

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

COMPARISON OF QUATERNION AND EULER ANGLE METHODS FOR JOINT ANGLE ANIMATION OF HUMAN FIGURE MODELS

by
Umit Y. Usta

March 1999

Thesis Co-Advisors:

Robert B. McGhee
Michael J. Zyda

19990406661
90406661
840

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

| | | |
|--|------------------------------|---|
| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE March 1999 | 3. REPORT TYPE AND DATES COVERED Master's Thesis |
| 4. TITLE AND SUBTITLE Comparison of Quaternion and Euler Angle Methods for Joint Angle Animation of Human Figure Models | | 5. FUNDING NUMBERS |
| 6. AUTHOR(S) Usta, Umit Y. | | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000 | | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) | | 10. SPONSORING / MONITORING AGENCY REPORT NUMBER |

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

| | |
|---|------------------------|
| 12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited. | 12b. DISTRIBUTION CODE |
|---|------------------------|

13. ABSTRACT (maximum 200 words)

This thesis presents articulated rigid body kinematics models for humans. The main area of research is to investigate models for real-time computer graphics applications in Virtual Environments (VE). Existing models have singularity problems and become too slow once the number of humans in view becomes large.

The approach taken is to develop a full body kinematics model with quaternions. Another common method, Euler angles, has singularity and interpolation problems. Both methods are compared for memory, computation and user input considerations. The implementation includes joint angle constraints. The model is then manipulated with user inputs by a mouse. As part of this research, the real-time display of human arm tracking with two inertial sensors, human walking, inverse kinematics, and key frame animation is also demonstrated.

The major conclusion of this thesis is that a kinematics model with quaternions can eliminate the singularity problems of existing models. Joint orientation interpolation is also more direct and less convoluted with quaternions. Neither representation exhibits a decisive advantage over the other in terms of computational speed. For memory considerations, the Euler angle method is best. To apply joint constraints, quaternion representations are converted to Euler angles, which causes additional computation for the system.

| | | | |
|---|--|---|----------------------------------|
| 14. SUBJECT TERMS virtual environment, articulated humans, human modeling, kinematics, sensors, postural control | | 15. NUMBER OF PAGES 223 | |
| 16. PRICE CODE | | | |
| 17. SECURITY CLASSIFICATION OF REPORT Unclassified | 18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified | 19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified | 20. LIMITATION OF ABSTRACT UL |

Approved for public release; distribution is unlimited.

**COMPARISON OF QUATERNION AND EULER ANGLE METHODS
FOR JOINT ANGLE ANIMATION OF HUMAN FIGURE MODELS**

Umit Y. Usta
Lieutenant JG., Turkish Navy
B.S.O.R. , Turkish Naval Academy, 1993

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

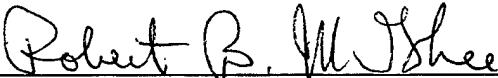
March 1999

Author:

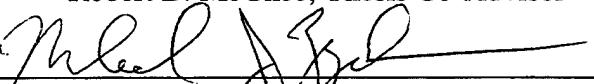


Umit Y. Usta

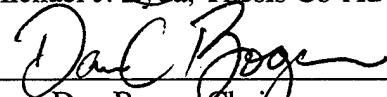
Approved by:



Robert B. McGhee, Thesis Co-Advisor



Michael J. Zyda, Thesis Co-Advisor



Dan Boger, Chairman
Department of Computer Science

ABSTRACT

This thesis presents articulated rigid body kinematics models for humans. The main area of research is to investigate models for real-time computer graphics applications in Virtual Environments (VE). Existing models have singularity problems and become too slow once the number of humans in view becomes large.

The approach taken is to develop a full body kinematics model with quaternions. Another common method, Euler angles, has singularity and interpolation problems. Both methods are compared for memory, computation and user input considerations. The implementation includes joint angle constraints. The model is then manipulated with user inputs by a mouse. As part of this research, the real-time display of human arm tracking with two inertial sensors, human walking, inverse kinematics, and key frame animation is also demonstrated.

The major conclusion of this thesis is that a kinematics model with quaternions can eliminate the singularity problems of existing models. Joint orientation interpolation is also more direct and less convoluted with quaternions. Neither representation exhibits a decisive advantage over the other in terms of computational speed. For memory considerations, the Euler angle method is best. To apply joint constraints, quaternion representations are converted to Euler angles, which causes additional computation for the system.

TABLE OF CONTENTS

| | | |
|-----|---|----|
| I. | INTRODUCTION | 1 |
| | A. MOTIVATION..... | 1 |
| | B. GOALS..... | 1 |
| | C. ORGANIZATION..... | 1 |
| II. | BACKGROUND | 3 |
| | A. GEOMETRY OF HUMAN MODELS (Modeling) | 4 |
| | 1. Skeleton Definition..... | 5 |
| | 2. Appearance..... | 7 |
| | a. Body Segments and Joints..... | 8 |
| | b. Clothes and Attached Objects..... | 9 |
| | c. Level of Detail..... | 9 |
| | B. BEHAVIOR OF HUMAN MODELS (Animation) | 10 |
| | 1. Transformation Hierarchy..... | 10 |
| | 2. Segment and Joint Deformation..... | 12 |
| | 3. Special Segments..... | 13 |
| | 4. Clothes and Attach Objects..... | 14 |
| | C. MANIPULATION OF HUMAN MODELS..... | 14 |
| | 1. Interactive Motion Control System..... | 15 |
| | 2. Scripting System..... | 17 |
| | 3. Hybrid Systems..... | 17 |

| | |
|--|----|
| D. SUMMARY..... | 18 |
| III. KINEMATIC MODELS..... | 19 |
| A. MDH NOTATION..... | 19 |
| B. JOINT TRANSFORMATION MATRIX..... | 25 |
| C. DISPLACEMENT ELEMENTS..... | 29 |
| D. SUMMARY..... | 30 |
| IV. COMPARISON OF QUATERNION AND EULER ANGLE MODELS..... | 33 |
| A. INTRODUCTION..... | 34 |
| B. VECTOR ROTATION..... | 35 |
| C. CONVERSION TO HOMOGENEOUS MATRIX..... | 41 |
| D. INTERPOLATION AND SINGULARITY..... | 45 |
| E. CONSTRAINT DEFINITION..... | 48 |
| F. HARDWARE, SOFTWARE AND NETWORK CONSIDERATION..... | 49 |
| G. USER INTERACTION..... | 50 |
| H. SUMMARY..... | 51 |
| V. FORWARD AND INVERSE KINEMATICS..... | 53 |
| A. DEFINITION..... | 53 |
| B. COMPUTATION..... | 54 |
| C. SENSOR PLACEMENT..... | 59 |
| D. SUMMARY..... | 61 |
| VI. IMPLEMENTATION AND RESULTS..... | 63 |
| A. HIERARCHY..... | 64 |
| B. USER INPUTS | 67 |

| | |
|--|-----|
| C. CONSTRAINTS..... | 70 |
| D. MOTION TRACKING..... | 70 |
| E. RESULTS..... | 72 |
| VII. SUMMARY AND CONCLUSION..... | 77 |
| A. SUMMARY..... | 77 |
| B. CONCLUSIONS AND FUTURE WORK..... | 78 |
| APPENDIX A: USER MANUAL..... | 81 |
| APPENDIX B: 3D HUMAN FIGURE SIMULATION SOFTWARE..... | 85 |
| LIST OF REFERENCES..... | 201 |
| INITIAL DISTRIBUTION LIST..... | 205 |

LIST OF FIGURES

| | |
|--|----|
| Figure 1 : Images from Geri's Game, 1997 (Pixar Animation Studios). [WEBREF1]..... | 5 |
| Figure 2 : Articulated Rigid Body Structure. | 6 |
| (a) Minimal Model (b) More Realistic Model | |
| Figure 3 : Position of Joints..... | 7 |
| Figure 4 : Handling Joint Deformation by Half-Angle Vertex Rotation | 13 |
| Figure 5 : IPORT Human Sensing Technology [SKOP96]..... | 16 |
| Figure 6 : Human Body Motion Tracking System from Polhemus. [WEBREF2]..... | 16 |
| Figure 7 : MDH Method Frame and Parameter Assignment [SKOP96]..... | 20 |
| Figure 8 : MDH Notation of Full Human Body Articulated Structure. | 22 |
| (a) Minimal Model (b) More Realistic Model | |
| Figure 9 : Rotation in 2D..... | 36 |
| Figure 10: Euler Angles..... | 37 |
| Figure 11: Vector-Angle Pair. (p is rotated on v by θ)..... | 37 |
| - Figure 12: Comparison of Methods for an Orientation..... | 43 |
| Figure 13: Comparison of Methods for A Rotation from Existing Orientation..... | 44 |
| Figure 14: Gimbal Lock (Airplane is attached to innermost ring) [WEBREF3]..... | 45 |
| Figure 15: Loop for Rigid Body Dynamics..... | 47 |
| Figure 16: A 2D Graphical Interaction Method for Vector-Angle Pair..... | 50 |
| Figure 17: Human Arm..... | 56 |
| Figure 18: A Minimally Sensed Human [SKOP96] | 60 |

| | |
|--|----|
| Figure 19: Proposed Hybrid Human Tracking Sensor Config. [FREY96]..... | 60 |
| Figure 20: Object Diagram of the Human Program..... | 64 |
| Figure 21: Segment Hierarchy..... | 65 |
| Figure 22: Drawing Left Arm..... | 66 |
| Figure 23: Draw Function of Segment Class | 66 |
| Figure 24: Calculating the Vector that is Perpendicular to the Screen..... | 68 |
| Figure 25: Response to Quaternion Input..... | 68 |
| Figure 26: Rotation of an Existing Orientation with a Vector-Angle Pair..... | 69 |
| Figure 27: Constraints for Quaternion Representation..... | 70 |
| Figure 28: Sensor Tracking Method..... | 71 |
| Figure 29: Construction of the Segment Transformation Matrix..... | 71 |
| Figure 30: Quaternion Interpolation | 72 |
| (a) User Defined Key Frames (b) Computed In-between Frames | |
| Figure 31: The Left Hand and the Left Foot Motions by Inverse Kinematics | 73 |
| Figure 32: Demonstration of Joint Constraints | 74 |
| (a) Impossible Motion for Elbow (b) Accepted Motion for Elbow | |
| Figure 33: Walking as a Procedural Animation..... | 74 |
| Figure 34: The Motion Tracking of Right Shoulder and Right Elbow with two Inertial Sensors and the Quaternion Attitude Filter..... | 75 |
| (a) Initial Posture (b) Initial Posture from another Point of View | |
| (c) Elbow Motion (d) Elbow and Shoulder motion | |
| (e) Shoulder has 90 degrees elevation (No singularity) | |

LIST OF TABLES

| | |
|---|----|
| Table 1. MDH Kinematics Parameters of Pelvic, Waist and Neck..... | 23 |
| Table 2. MDH Kinematics Parameters of Shoulders..... | 23 |
| Table 3. MDH Kinematics Parameters of Elbows and Wrists..... | 24 |
| Table 4. MDH Kinematics Parameters of Hips..... | 24 |
| Table 5. MDH Kinematics Parameters of Knees and Ankles..... | 25 |

ACKNOWLEDGMENTS

Many thanks to all those whose helped make this thesis possible. Special thanks to my two thesis advisors. My sincerest thanks to Dr. Robert McGhee for his patience, encouragement, and devotion to his students. His vast experience and unbounded enthusiasm have made working with him a true delight. To Dr. Michael Zyda I owe much for his guidance in Computer Graphics. I would also like to thank to Eric Bachmann and John Falby for their many hours of instruction over the past two years that have brought me a long way in understanding the science of computers.. Finally many thanks to all members of NPSNET Research Group and to all of the faculty, students and staff of the Computer Science Department who helped in numerous ways.

I. INTRODUCTION

A. MOTIVATION

There is a growing requirement for realistic virtual environments (VE) in which humans can interact. Recent advances in computer and motion sensor technologies have made it feasible to insert humans into the VE and permit them to interact with their environments. For realism, one of the major requirements is that the response time of the simulated human model must be real time and the motion must be smooth. The motions of the human are represented in the model with transformation of body parts. One of the most popular representations of a transformation is to use Euler angles. While this is easy to understand and use, it has singularity problems, which causes unrealistic motion and divide-by-zero errors in the system. An alternative method is the use of unit quaternions. The quaternion method experiences no singularities at any orientation. The interpolation is also more direct and less convoluted with quaternions.

B. GOALS

The purpose of this thesis is to compare two different methods used for transformation matrices and model the human body with quaternions. The model manipulation is demonstrated by user inputs with a mouse. The singularity problems are examined for both methods. For realism, joint angle limits are added to the model. This thesis also demonstrates the real-time display of human arm tracking with two inertial sensors.

C. ORGANIZATION

Chapter II of this thesis provides background information regarding human models. Chapter III provides an overview of kinematics models and discusses joint transformation

matrices. Chapter IV compares two methods used to construct joint transformation matrices. Chapter V introduces forward and inverse kinematics and makes a comparison of their computational speed and input requirements. Chapter VI contains the implementation details and presents results obtained from this research. The last chapter, Chapter VII, provides some conclusions and discusses recommendations for future enhancements and research relating to the work of this thesis.

II. BACKGROUND

Research in three fields is relevant to the problem of animating human motion: robotics, biomechanics, and computer graphics [HODG95]. Today many applications in these fields use human figures or basic principles that can be used to design control strategies for humanlike models. Walking machines [MCGH86], human figure simulation programs [BADL93a], and character animation tools [WAVE98] are some of these applications.

Today, many tasks can be accomplished by using computer graphic applications that feature human figures. Human factors design engineers or ergonomics analysts can study, analyze, assess, and visualize human motor performance, fit, reach, view, and other physical tasks in a workplace environment by using computer simulated humans in the early design stages [BADL93a]. Animators can create human characters that can walk, jump, and even dance with incredible realism for games, cartoons, video-clips, movies and advertisements by using character animation tools. The representation of the human figure in a real-time interactive 3D virtual environment (VE) is a long sought for goal of the VE research community [DURL95]. Simulated autonomous human agents are needed in VE application areas such as training, education, and entertainment. Human motion capture systems, which help to insert individual users in VE, also use mathematical models of human figures. All these applications have different requirements for computation speed, appearance realism, motion realism, and usability of human figures.

This chapter provides background information regarding computer representation of human figures. While human figures are introduced under modeling, animation and interaction sub topics, none of them can be separated from each other and all are affected

by the trade-off between realism and computation speed. Because the main area of this research is to investigate human figures for real-time computer graphic applications in VE, all issues of human figure representation and modeling are examined for computation speed. Currently available realistic models are also briefly introduced, but the focus is on the skeletal system of the human body.

A. GEOMETRY OF HUMAN MODELS (Modeling)

For realism, one would expect a human model to be structured like the human skeletal system, to have a humanlike appearance, and to be sized according to permissible human dimensions. Appearance involves a compromise between realism and display speed. No one is likely to mistake the figure for a real person; on the other hand, the movements and the speed of control are good enough to convey a suitably responsive attitude [BADL93a].

Today it is possible to create human models that have a realistic appearance and motion. These models also have skin and muscle animation. Their hair and clothes are animated as they move in real world. They can talk, look around, and make facial expressions. They can grasp or make a gesture with their hands. But they can not do all of these in real-time. Figure 1 is an example of realistic human figures. While making a geometrical definition of the model, it is also necessary to consider animation issues like manipulation and deformation algorithms and their input parameters. Decisions regarding animation techniques effects the modeling phase and many animation techniques that are used for realism are not applicable for real-time systems.

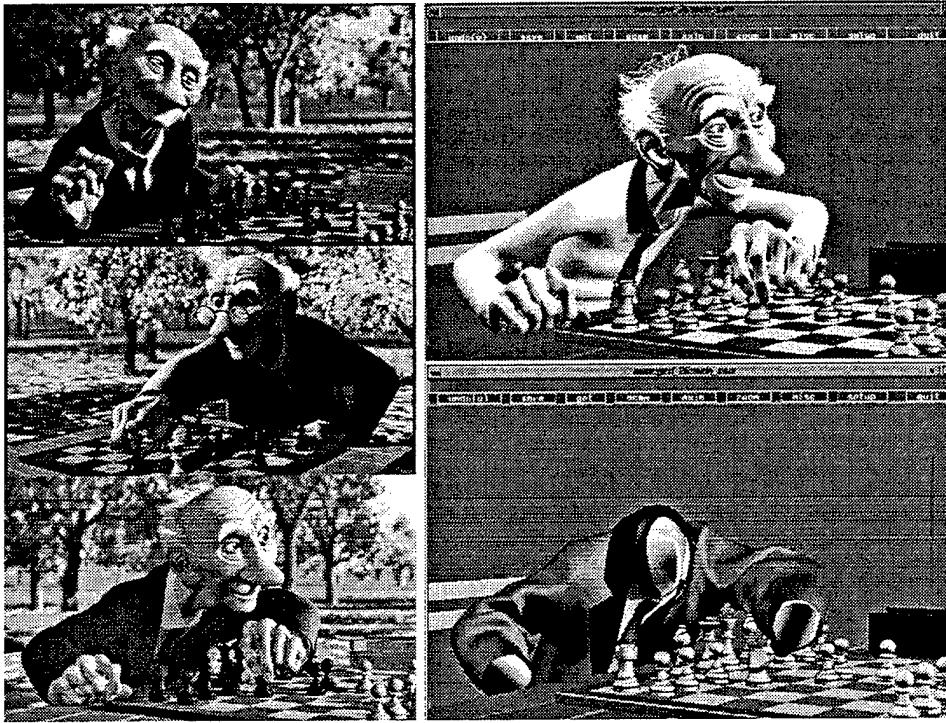


Figure 1: Images from Geri's Game, 1997 (Pixar Animation Studios). [WEBREF1]

1. Skeleton Definition

Most animation of clothed structures is controlled by animating an underlying skeleton of some description and then rendering the final images with flesh or clothes [WATT92]. A representation of the skeleton under the skin of a human model amounts to an *articulated rigid body* [CRAI89].

An articulated structure is made up of connected segments that move relative to each other for the definition of the human posture. Each link, or human link segment, has its physical dimensions. Links are connected each other with joints, and constitute a composition hierarchy that has a tree structure. Each joint angle defines an orientation for the outboard links. The number of independent position variables necessary to specify the state of the joint is called its “degrees of freedom” (DOF). The DOF of an articulated

structure is the total DOF of all joints. Each link motion, joint displacements and rotations, is also carried to child links.

The hierarchy between rigid body segments changes when the root link of the tree is changed. Changing the root link does not effect the relation between links; it only effects the transformation hierarchy of segments. That is why leaf nodes of the hierarchy tree never change. Leaf nodes are hands, feet and head, also known as “end-effectors” [CRAI89]. The tree structure for the model used in this thesis is shown in Figure 2.

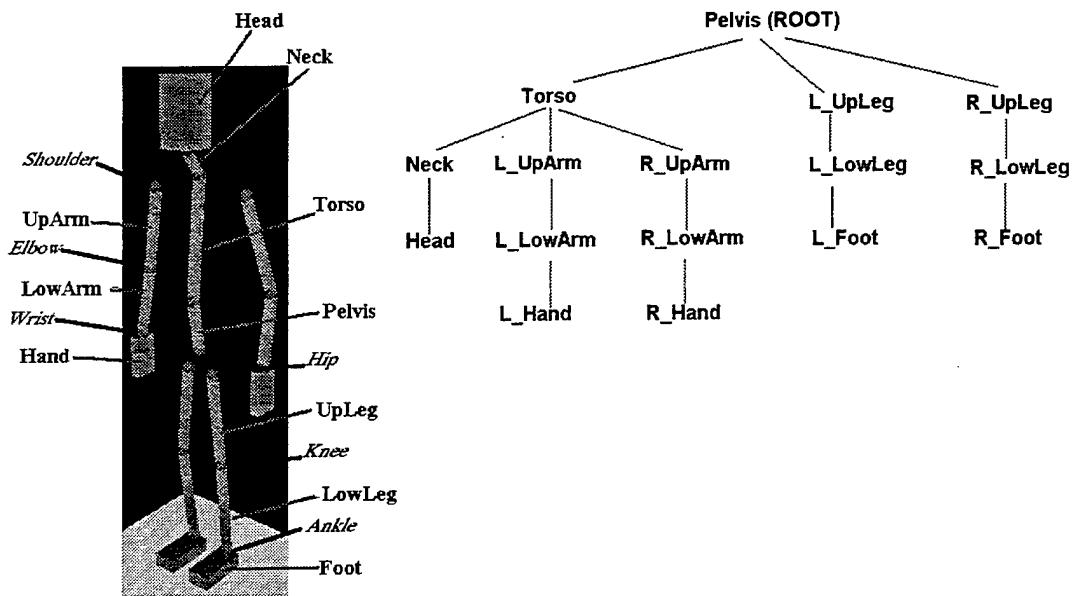


Figure 2: Articulated Rigid Body Structure

While there are over 200 joints in the human skeletal system, it is unnecessary and impractical to model each segment as an articulated rigid body. The number of human segments must be chosen to supply both realism and sufficient computation speed. Toward this end, it can be noted that there are mainly 15 major body segments in the human body; namely: head, torso, pelvis, upper arms, forearms, hands, upper legs, lower

legs, and feet. A neck can be added as 16th segment. Each added segment supplies more realism, while decreasing computation speed. Figure 3 demonstrates two different articulated bodies. Figure 3a has minimal 15 joints and 34 DOF. Figure 3b adds 5 more joints and 11 more DOF, while representing the torso by a spine.

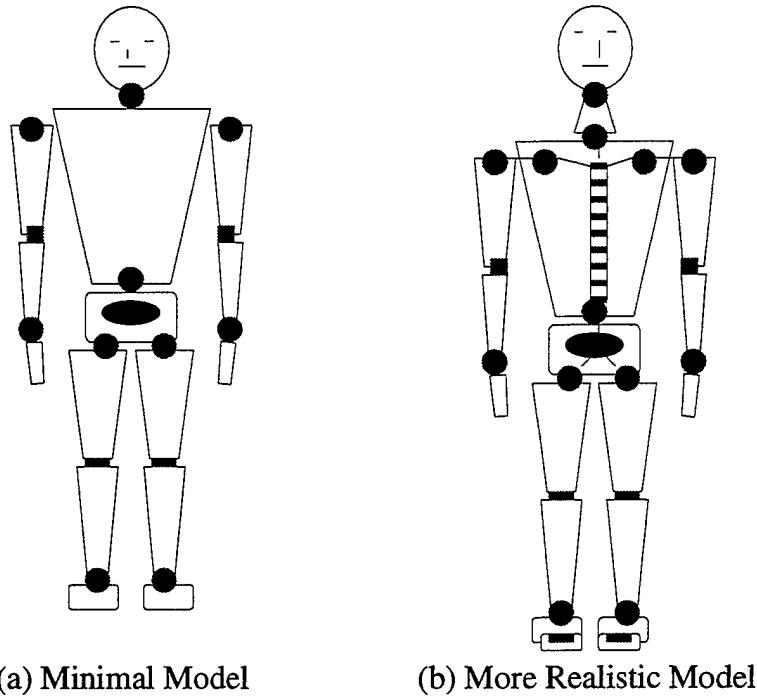


Figure 3: Position of Joints

In some applications; it is not possible to treat some special segments as rigid bodies. These segments are the head, torso, hands and feet. Hands have their own articulated rigid bodies. Feet can be modeled as two articulated rigid bodies (toe and heel). Eyes are articulated bodies of the head. The torso can be defined as a spine.

2. Appearance

The number of human link segments and their sizes effect the realism of a graphics model. In addition, each human segment has its own shape, and all have the same geometrical definition. While shapes are the main concern for appearance, their

geometrical definition is important for animation. The realistic appearance of the human segment shapes or whole body figure may be lost once animation is started. Geometrical definition of the shape is very important for the trade-off between realism and computation speed.

a. Body Segments and Joints

Segment shapes may be sticks, naked skin, or parts of clothing. Segment geometry may be surface models, volume models, or solid geometry models.

Surface models are polygons and patches. Polygon models are relatively simple to define, manipulate, and display. They are the most common models processed by graphics workstation hardware and commercial graphics software. All viable interactive human figure models are done with polygons. Several hundred polygons can look acceptably human like; accurate skin models require thousands of polygons [BADL93a]. Joints may be drawn by overlapping segments or by using patches. Patches are curved surfaces and are usually defined by cubic polynomials. Control points and tangent vectors define the shape of the patch. For the human figures that are modeled by patches, each human segment has its own control points. Adjacent segments share the control points of a joint patch. While patches add realism at joints for smoothness and proper bending, they are not as yet applicable for real-time systems.

Volume models or solid geometry models are composed of non-intersecting elements within a spatial partition, such as voxels or oct-trees, or created from (possibly overlapping) combination of inherently 3D primitive volumes [BADL93a]. Voxel models are used for modeling the anatomic structure of a human body, but not for modeling human figures. Primitive solid geometry models are very old and unrealistic.

An interesting generalization of spheres (metaballs) is a potential function with a center and a field function that decreases monotonically from the center outward. Metaballs were originally used to model molecules. They represent a very slow, but an interesting possibility for highly realistic models in the future [BADL93a].

The head is a special segment and needs a combination of different types of geometric models. For facial animation, the face should be modeled as patches. Hair may be a fuzzy object. Segment shapes are skin or parts of clothing for human figures used for real-time computer graphic applications in VE. Geometrical definitions are polygons and segments overlap at joints, so the head doesn't present any special concerns.

b. Clothes and Attached Objects

It is possible to attach some objects to some body segments. Attached objects will behave as a new articulated body. They may have their own geometric definition and animation. They may be attached or detached dynamically at run-time. Clothes may be defined as attached objects, which necessitates time-consuming algorithms. For real-time computer graphic applications, clothes are treated as body segment shapes and the geometry of all attached objects is polygon. They may be textured for realism.

c. Level of Detail (LOD)

Multi-resolution models are very important for real-time systems. In such an approach, model details and size reduce with increasing distance from the observer. Since finer details are less pronounced, they need not be rendered and may be left out of the model thereby putting fewer polygons into the graphics pipeline, allowing for higher frame rates. It is important to note that when polygons transform to less than one pixel, they effectively combine [PRAT93]. As distances increase, this natural occurrence aids

in reducing visual detail and supports the use of LOD models. For low-resolution human models, the number of links needed for skeletal representation and the number of polygons needed for segment shapes are reduced. Animation algorithms that define some procedural motions like walking may be simplified for low-resolution models.

B. BEHAVIOR OF HUMAN MODELS (Animation)

For realism, a human model should move or respond like a human and should exist, work, act and react within a three-dimension VE [BADL93a]. The main animation of the human model is to create posture, which is controlled by the articulated skeletal structure. Input parameters of the system are the DOF of this structure. By defining simple sets of rules for how segments behave, input parameters may be reduced for the user. This is also needed for realistic behavior of skeletons. Constraints and control mechanisms may warn users of unacceptable inputs. Some motions may be generated automatically to simplify user control. After defining articulated structure animation, secondary animations like muscle and clothing may be added for realism. Facial and hand animations are other special topics for realistic human figures.

1. Transformation Hierarchy

The main concern for human posture modeling is the skeleton, its articulated structure. The articulated structure represents tree-structured human segments through a hierarchy and defines the position and orientation of each human segment. Link motions have constraints, depending on the model definition.

A kinematics model specifies motion independent of the underlying forces, which defines geometrical and time related properties of motion, such as position, velocity and acceleration of each link. A kinematics model also sets the positional and angular

constraints of the human segments. For forward kinematics, all transformations are specified to control the motion of the end-effectors. For inverse kinematics, a goal is specified for the end-effector and the system computes the transformations required to achieve the goal. It is possible to think of inverse kinematics as a numerical engine that can be attached to any part of the skeleton whose purpose is to specify the position and orientation of all the nodes between the end node and base node [WATT92]. It is also possible to apply more than one type of model to the same skeletal structure. For example, while the torso and head use forward kinematics, arms and legs could use inverse kinematics, where hands and feet are end-effectors, and shoulders and hips are base nodes. For combined models, base and root nodes may be redefined at run time. Redefining the skeletal hierarchy at run time adds more control over human segments, while decreasing computation speed. One example of changing the root node at run time is the animation of legs from the hip down during the transfer phase of the walking cycle and from the foot up during the support phase, which prevents the collision between foot and the floor.

A dynamic model specifies motion taking into account physical attributes, properties, and laws. It introduces physical properties, such as mass and moments of inertia. It is possible to simulate human motion realistically with detailed dynamic models. However, the cost of this realism is a high degree of computational complexity. When more detailed models are chosen, the response time of the simulated human model increases. A more realistic approach may be to consider the connection between limb segments as not rigid [BEDI97]. In this approach, joints behave as springs. A dynamic model may also be implemented as a forward or inverse dynamics system. For inverse dynamics, the motion

of each segment is given and the forces and torque are computed. From this, a direct dynamics model may be derived [KOOZ83]

In computer graphics, joints are usually defined as rotary (revolute) joints, although there exist other types, such as prismatic (sliding) joints. Mathematical models of joints should supply the necessary transformations. The most popular representation of spatial transformations of point vectors is the 4x4 real matrix (also termed homogeneous transform), based on the idea of homogeneous coordinates. The appeal of homogeneous transforms is mostly due to their matrix manipulation by a computer. On the other hand, such matrices are highly redundant, using 16 numbers (of which four are trivial) to represent rotation and position. This redundancy can introduce numerical problems in calculations, wastes storage, and often increases the computational cost of algorithms. Despite these drawbacks, matrix-based representations remain the dominant choice for most robotic system applications [FUND90]. Two types of methods can define the rotation part of the matrix: Euler angles and angle/vector pair. An alternative representation for a vector-angle pair is the quaternion. Each method has pros and cons. The quaternion method is the only one that can rotate vectors without using matrix multiplication, which can eliminate the need of generating and applying homogeneous transform matrices. The following chapters of this thesis will focus on comparison of quaternion and Euler angle methods.

2. Segment and Joint Deformation

Segments may have their own animation. Muscles can be treated as soft objects and animated by free form deformation (FFD) blocks, which adds additional computation. Joint deformation can be handled by using patches. Another way is the using polygon

subdivision algorithm for the joints. The last adds more realism and solves the continuity problem of patches. If the geometry of segments is a “metaball”, the joint surface problem is solved in its definition. Segment and joint deformations are not applicable for human figures in today’s real-time VE. Figure 4 demonstrates a joint deformation, which rotates waist vertices through the half angle of the orientation of outboard torso segment and scales the same vertices relative to the orientation. This solution is simple, and works well for small joint motions, but produces unrealistic results for large joint motions. To the author’s knowledge, there is, as yet, no satisfactory real-time algorithm for realistic representation of joint deformation in human figure models.

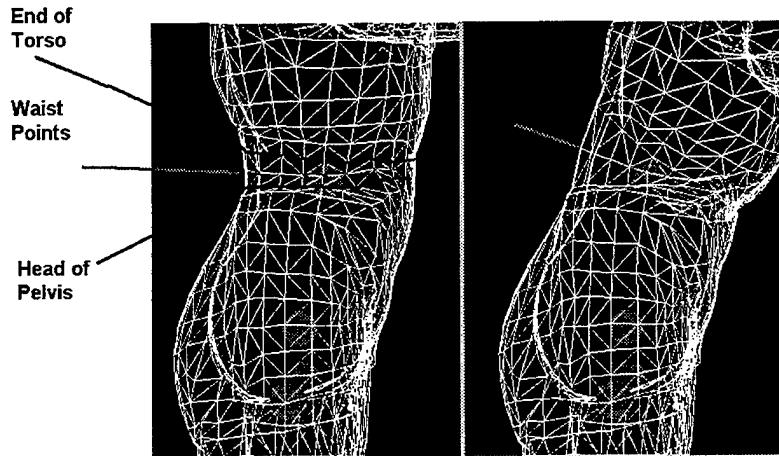


Figure 4: Handling Joint Deformation by Half-Angle Vertex Rotation

3. Special Segments

Torso and end-effectors need special animation techniques. It is possible to treat all as single rigid bodies for real-time systems. Some real-time systems that use the Cyber-Glove as an interaction device need hand models and animation algorithms. Non-real

time applications like character animation tools have special models and animation techniques for these segments.

To achieve a higher degree of realism, the torso needs to be treated as supported by a flexible spine rather than a rigid body. The head needs hair motion and facial animation. The face needs eye motion, lips and skin motion. The face usually uses a parametric muscle model that controls the deformation of a polygonal mesh representation. Hands are also complex objects. There are many links in a hand and each must react realistically with each other. Accurate hand simulation needs grasping algorithms depending on the target object and the grasp type. A foot model could supply toe and heel articulation.

4. Clothes and Attached Objects

If clothes are defined as attached objects, they may be animated independent of, but constrained by the segments. Dynamic behavior and collision detection algorithms must be defined for the attached clothing. Clothing also constrains movement by effecting joint angle limits. For real-time simulation, attached clothes like a hat or glasses and other attached objects like rifle or bag are always static objects. They don't have their own animation, and do not effect joint angle limits at run-time while they may be attached or detached at run-time.

C. MANIPULATION OF HUMAN MODELS

An interactive software tool must be designed for usability. Existing interaction paradigms (such as pop-up menus or command line completions) should be followed when they are the most efficacious for a particular task, but new techniques will be needed to manage and control three-dimensional articulated structures with standard graphical input tools [BADL93a].

A high-level animation system allows the animator to specify the motion in abstract general terms, whereas a low-level system requires the animator to specify individual motion parameters manually. High-level commands describe behavior implicitly in terms of events and relationships, whereas lower level commands are far more explicit [WATT92]. The purpose of high level control is to reduce the number of control parameters for the system and leave those low-level parameters to the computer to generate. For example, inverse kinematics for hand motion is high-level motion control. Animation systems may use some forms of procedural animation, like walking, running, jumping, grasping, bending, facial expressions, talking and many others for high-level motion control systems. Here, motion is described by a mathematical model/algorithm. These animations can take place with minimum inputs for real-time systems. For example, in the Individual Soldier Mobility System, a soldier sits in a room called the Walk-in Synthetic Environment on a pedal-based mobility simulator. The soldier moves through the environment by pedaling. Pedaling speed is used by the “Jack” model to provide realistic joint angles for lower body [BADL93a]. The lower body is rendered as standing, walking and running. The I-PORT system component is shown in Figure 5.

On the other hand, human body motion tracking systems provide low-level control for each joint, which defines more realistic motions for each user in real-time. Figure 6 shows a human body motion tracking system.

1. Interactive Motion Control System

A simple interactive motion control system allows a user to set up a sequence by interactively specifying a path and kinetic characteristics, using two-dimensional interactive graphics devices. Other expensive interactive motion control systems use

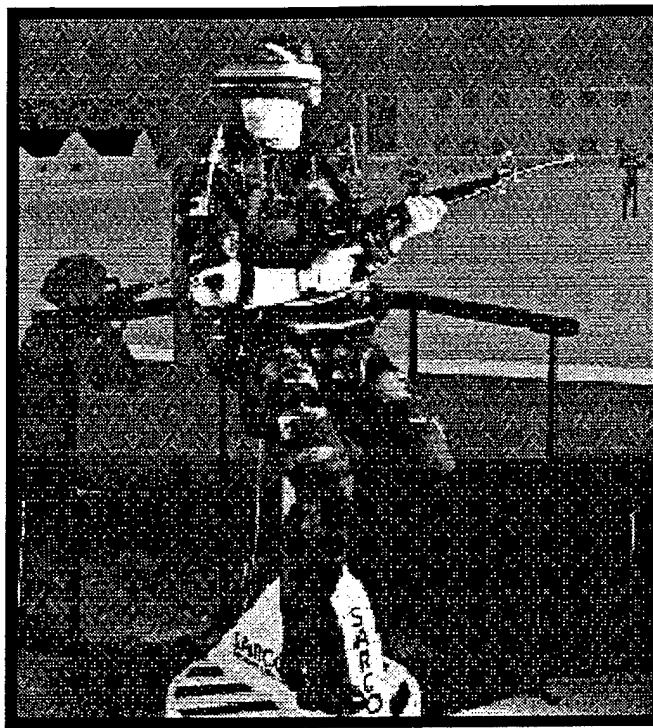


Figure 5: IPORT Human Sensing Technology [SKOP96]



Figure 6: Human Body Motion Tracking System from Polhemus. [WEBREF2]

motion tracking devices. Currently, there are five types of motion trackers: mechanical, acoustic, optical, electromagnetic and inertial trackers. All have pros and cons. Optical and electromagnetic trackers are mostly used for motion tracking of body segments. Currently, inertial trackers are being developed to track human body segments [DUMA99]. Optical trackers are also popular for tracking facial expressions. Different types of Cyber-Gloves are used for hand motion tracking.

For human body motion tracking systems, control parameters must be checked for validity by using human model constraints. For example, the Jack model checks upper body angles in the Individual Soldier Mobility System. If the sensors track only the end-effector's motion, a human model must control reachable space and compute motion of in-between links by using inverse kinematics or inverse dynamics.

2. Scripting System

For character animation, a user can write a script to define the motion. This is the earliest type of motion control systems. For a scripting system, the user needs skill in the language. The advantage is the definition of high-level motions to create motion libraries. Most of the animations today are done on an interactive system rather than scripting systems, which supports real-time animation.

3. Hybrid Systems

There are some systems that use both control types. Today, most well known character animation tools use hybrid systems. Some procedural motions may be defined by using motion capture and key-frame animation. Motion libraries can be created for each procedural motion or combination of more than one procedural motion.

Accumulated expertise is an advantage in the use of scripted languages to edit a sequence, to build up libraries and to approach more and more complex problems.

D. SUMMARY

There are many issues for modeling human figures. New methods are developing for more realism and automatic control of the figures. But, the real-time requirement usually constrains the animation to consist of flat or Gouraud-shaded polygons with texture mapping [WATT92]. For real-time systems, joint motions still must be realistic, after making a high level of abstraction for the appearance. Articulated structure is the most important part of the human figure to define realistic postures and human segment motions. Computation using this structure is demanding for real-time systems. Redefining structure at run time adds more control over human segments, while decreasing computation speed. If the real-time system is a networked synthetic environment, the state vector of the articulated structure should be minimized for efficient network traffic. The following chapters will focus on efficient articulated structure representations and the comparison of quaternion and Euler angle methods that construct joint transformation matrices. An articulated rigid body with quaternions is introduced in Chapter IV.

III. KINEMATIC MODELS

Kinematics models specify motion independent of the underlying forces, which defines geometrical and time related properties of motion, such as position, velocity and acceleration of each link. For forward kinematics, all joints are specified explicitly by the animator. The motion of the end-effector is determined indirectly as the accumulation of the transformations that lead to that end-effector, as the tree of the structure is descended. For inverse kinematics, sometimes called “goal directed” motion, the animators define the end-effector only. Inverse kinematics solves for the position and the orientation of all joints in the link hierarchy that leads to the end-effector [WATT92]. Usually, forward kinematics is used to render predefined postures, while inverse kinematics is used for processing motion tracker data and motions paths of end-effectors. These two types of kinematics are detailed in Chapter V. In this chapter, kinematics notation for the human body articulated structure and transformation matrices are discussed.

A. MDH NOTATION

Two common methods to represent an articulated figure mathematically are the Danevit-Hartenberg (DH) notation and the Craig notation that is also known as the modified DH method (MDH). These methods were originally developed for robotic manipulations. Both describe the kinematics of each link relative to its neighbors by attaching a coordinate frame to each link. Each joint has 1 DOF. The MDH method attaches the coordinate origin for each link to its inboard joint motion axis while the DH method attaches the origin to the link’s outboard motion axis. The base joint is numbered as joint 0 for both methods. Numbering is always increase from root link to outward [CRAI89].

Links and the link's inboard joints have the same index number for the MDH notation. Figure 7 shows MDH method frame and the parameter assignments. The Z-axis is coincident with the joint motion axis. The X-axis that is the common normal of the link and is directed from the link's inboard joint towards it's outboard joint. The X-axis intersects both joint axes at right angles. The Y-axis completes a right hand orthogonal set and is needed only for specifying the 3D shape of a link.

Four parameters are needed to describe the relation of two consequent frames. Link length is the distance along the X-axis between the joints of a given link. Link twist is the angle between inboard joint axis and outboard joint axis measured about the X-axis. Link offset is the distance measured at the inboard link motion axis from the preceding X-axis to the current link X-axis. Joint angle is the rotation measured at inboard joint motion axis from the previous link X-axis to the current link X-axis.

- inboard link length

a_{i-1} = distance from \hat{z}_{i-1} to \hat{z}_i measured along \hat{x}_{i-1}

- inboard link twist

α_{i-1} = angle between \hat{z}_{i-1} and \hat{z}_i measured about \hat{x}_{i-1}

- outboard link offset

d_i = distance from \hat{x}_{i-1} to \hat{x}_i measured along \hat{z}_i

- outboard joint angle

Θ_i = angle between \hat{x}_{i-1} to \hat{x}_i measured about \hat{z}_i

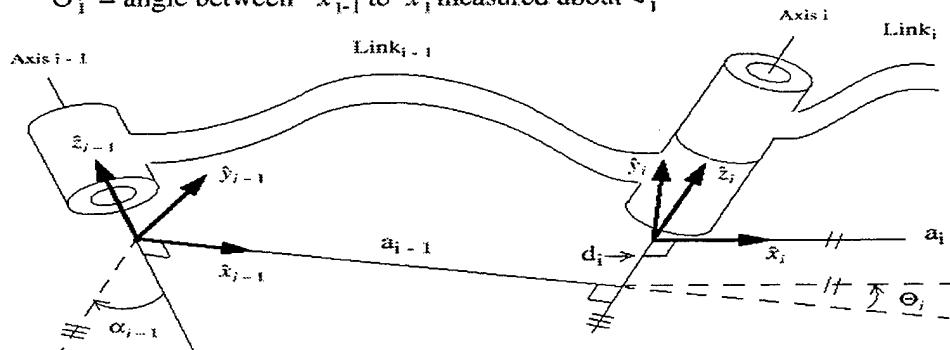


Figure 7: MDH Method Frame and Parameter Assignment [SKOP96]

Outboard joint position and orientation relative to inboard joint can be computed by these four parameters. Four steps of this computation are rotation about X-axis as inboard link twist, displacement along X-axis as inboard link length, rotation about outboard joint axis as outboard joint angle and displacement along outboard joint axis as outboard link offset. All these rotations and displacements are represented by individual homogeneous transformation matrices. On the other hand, a single transformation matrix can define all four motions by multiplying these four transformation matrices in the specific order with the following result.

$$T = Rx(\alpha_{i-1}) Dx(a_{i-1}) Rz(\Theta_i) Dz(d_i) \quad (3.1)$$

This produces the matrix:

$$T = \begin{bmatrix} \cos\Theta_i & -\sin\Theta_i & 0 & a_{i-1} \\ \sin\Theta_i \cos(\alpha_{i-1}) & \cos\Theta_i \cos(\alpha_{i-1}) & -\sin(\alpha_{i-1}) & -\sin(\alpha_{i-1}) d_i \\ \sin\Theta_i \sin(\alpha_{i-1}) & \cos\Theta_i \sin(\alpha_{i-1}) & \cos(\alpha_{i-1}) & \cos(\alpha_{i-1}) d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.2)$$

The last row of the matrix given by Eq.(3.2) is redundant and contains no information about the relation. First three elements at the last column represent the local displacements. Other nine elements that constitute a 3x3-matrix represent the rotation. Later sections in this chapter focus on this matrix. This notation is used to describe linked structures where the joints have a single DOF.

Ball joints can be represented as multiple single DOF joints located at the same point in space [WATT92]. Figure 8 represents a full human body articulated structure defined by MDH. Only the Z-axes of MDH frames are shown. Joints that have more than 1 DOF have more than one Z-axis originating at the same point, where in-between link offsets and link lengths are all zero. The root segment is the pelvis, and the first frame is the

twist axis of this segment. The standard MDH numbering system differs at branching points of the tree structure.

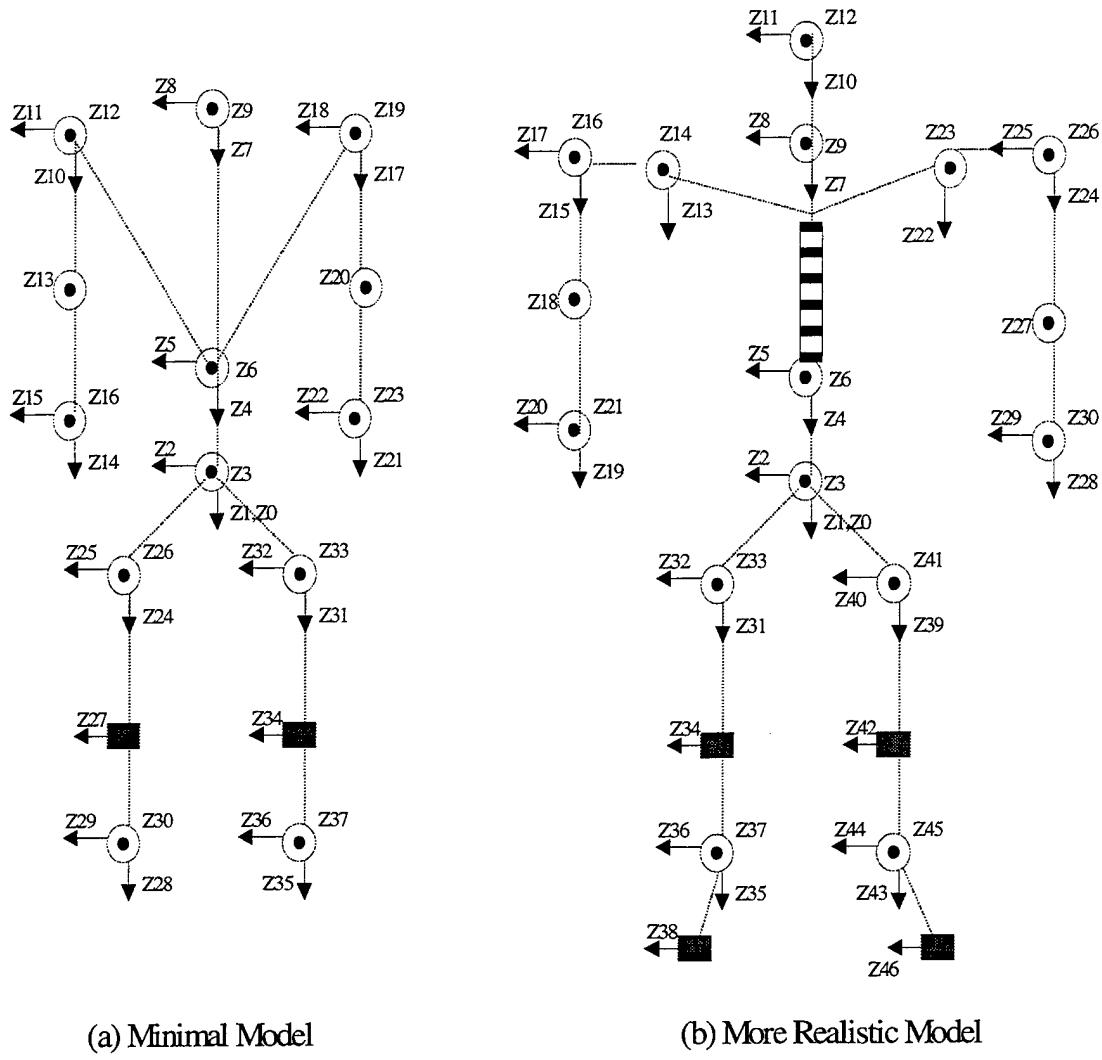
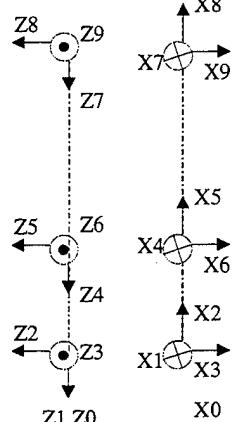


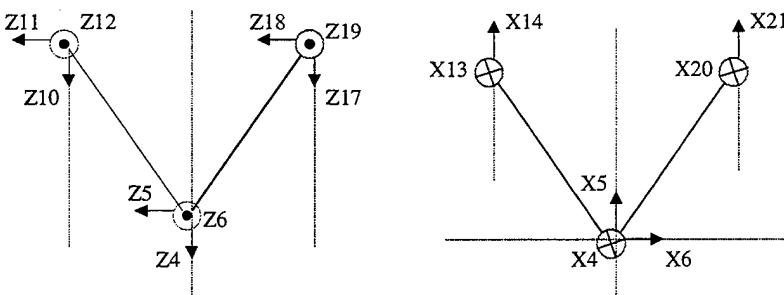
Figure 8: MDH Notation of Full Human Body Articulated Structure

Table 1-5 provides kinematic parameters of the minimal model that also has a neck joint. The frame's X-axes are also drawn to determine the sign of the twist angles. Notice that, in this notation, knees and elbows have 1 DOF and all other joints have 3 DOF. Joint limits are also defined, but they will be discussed in the next chapter.



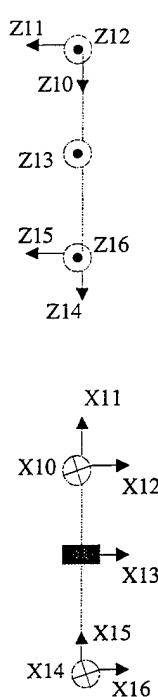
| | Inboard link twist | Inboard link length | Outboard link offset | Outboard link angle | Θ_i | Θ_i |
|---------------|--------------------|---------------------|----------------------|---------------------|------------|------------|
| | ∞_{i-1} | a_{i-1} | d_i | Θ_i | MIN | MAX |
| Pelvic | ∞_{i-1} | a_{i-1} | d_i | Θ_i | - | - |
| | Rz | 1 | 0 | 0 | 0 | - |
| | Ry | 2 | 90 | 0 | 0 | - |
| Waist | Rz | 3 | 90 | 0 | 0 | - |
| | <u>twist</u> | 4 | 90 | 0 | Waist_Y | 0 |
| | <u>lean</u> | 5 | 90 | 0 | Waist_X | 0 |
| Neck | <u>bow</u> | 6 | 90 | 0 | Waist_Z | 0 |
| | <u>twist</u> | 7 | 90 | 0 | Neck_Y | 0 |
| | <u>lean</u> | 8 | 90 | 0 | Neck_X | 0 |
| | <u>bow</u> | 9 | 90 | 0 | Neck_Z | 0 |

Table 1. MDH Kinematics Parameters of Pelvic, Waist and Neck



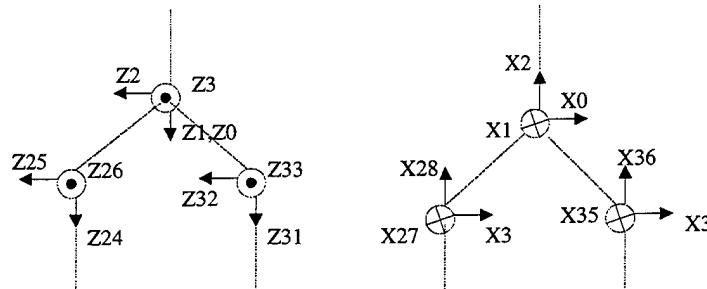
| | Inboard link twist | Inboard link length | Outboard link offset | Outboard link angle | Θ_i | Θ_i |
|-----------------------|--------------------|---------------------|----------------------|---------------------|------------|------------|
| | ∞_{i-1} | a_{i-1} | d_i | Θ_i | MIN | MAX |
| LEFT Shoulder | INDEX | ∞_{i-1} | a_{i-1} | d_i | Θ_i | Θ_i |
| | <u>twist</u> | 10 | 90 | Shoulder_X | Shoulder_Y | 0 |
| | <u>swing</u> | 11 | 90 | 0 | 0 | -80 180 |
| RIGHT Shoulder | <u>roll</u> | 12 | 90 | 0 | Shoulder_Z | 0 |
| | <u>twist</u> | 19 | 90 | Shoulder_X | Shoulder_Y | 0 |
| | <u>swing</u> | 20 | 90 | 0 | 0 | -80 180 |
| | <u>roll</u> | 21 | 90 | 0 | Shoulder_Z | 0 |
| | | | | | -30 | 180 |

Table 2. MDH Kinematics Parameters of Shoulders



| | Inboard link twist | Inboard link length | Outboard link offset | Outboard link angle | Θ_i | Θ_i |
|----------------|--------------------|---------------------|----------------------|---------------------|------------|------------|
| | ∞_{i-1} | a_{i-1} | d_i | Θ_i | MIN | MAX |
| L_Elbow | ∞_{i-1} | a_{i-1} | d_i | Θ_i | MIN | MAX |
| roll | 13 | 0 | upArmLength | 0 | 0 | 100 |
| R_Elbow | ∞_{i-1} | a_{i-1} | d_i | Θ_i | MIN | MAX |
| roll | 20 | 0 | upArmLength | 0 | -100 | 0 |
| L_Wrist | ∞_{i-1} | a_{i-1} | d_i | Θ_i | MIN | MAX |
| twist | 14 | 90 | 0 | Wrist_Y | 0 | -20 20 |
| swing | 15 | 90 | 0 | Wrist_X | 0 | -20 20 |
| roll | 16 | 90 | 0 | Wrist_Z | 0 | -30 70 |
| R_Wrist | ∞_{i-1} | a_{i-1} | d_i | Θ_i | MIN | MAX |
| twist | 21 | 90 | 0 | Wrist_Y | 0 | -20 20 |
| swing | 22 | 90 | 0 | Wrist_X | 0 | -20 20 |
| roll | 23 | 90 | 0 | Wrist_Z | 0 | -70 30 |

Table 3. MDH Kinematics Parameters of Elbows and Wrists



| | INDEX | ∞_{i-1} | a_{i-1} | d_i | Θ_i | Θ_i |
|------------------|-------|----------------|-----------|-------|------------|------------|
| | | | | | MIN | MAX |
| LEFT Hip | | | | | | |
| twist | 24 | 90 | -Hip_X | Hip_Y | 0 | -30 30 |
| swing | 25 | 90 | 0 | 0 | 0 | -50 100 |
| roll | 26 | 90 | 0 | Hip_Z | 0 | -90 10 |
| RIGHT Hip | | | | | | |
| twist | 31 | 90 | Hip_X | Hip_Y | 0 | -30 30 |
| swing | 32 | 90 | 0 | 0 | 0 | -50 100 |
| roll | 33 | 90 | 0 | Hip_Z | 0 | -10 90 |

Table 4. MDH Kinematics Parameters of Hips

| | Inboard link twist | Inboard link length | Outboard link offset | Outboard link angle | Θ_i | Θ_{MIN} | Θ_{MAX} |
|----------------|--------------------|---------------------|----------------------|---------------------|------------|----------------|----------------|
| <u>L_Knee</u> | ∞_{i-1} | a_{i-1} | d_i | Θ_i | | | |
| <u>swing</u> | 27 | 90 | 0 | HipLength | 0 | -100 | 0 |
| <u>R_Knee</u> | | | | | | | |
| <u>swing</u> | 34 | 90 | 0 | HipLength | 0 | -100 | 0 |
| <u>L_Ankle</u> | | | | | | | |
| <u>twist</u> | 28 | 90 | 0 | Leg_Y | 0 | -20 | 20 |
| <u>swing</u> | 29 | 90 | 0 | Leg_X | 0 | -70 | 30 |
| <u>roll</u> | 30 | 90 | 0 | Leg_Z | 0 | -20 | 20 |
| <u>R_Ankle</u> | | | | | | | |
| <u>twist</u> | 35 | 90 | 0 | Leg_Y | 0 | -20 | 20 |
| <u>swing</u> | 36 | 90 | 0 | Leg_X | 0 | -70 | 30 |
| <u>roll</u> | 37 | 90 | 0 | Leg_Z | 0 | -20 | 20 |

Table 5. MDH Kinematics Parameters of Knees and Ankles

B. JOINT TRANSFORMATION MATRIX

All joints in the human articulated structure are revolute rigid joints. That is why the inboard link length, inboard link twist, and outboard link offset are constant values. They can be set at the construction phase of the human figure. The only variable is the outboard joint angle. Because of the fact that two displacement and one rotation matrices don't change at run-time, constructing a transformation matrix for MDH notation may be an advantage against making unnecessary four matrix multiplications at run time.

Another advantage occurs when a twist angle is defined as a right angle or a zero angle since either the sine or cosine of the angle will be zero in this case. For the joints that have more than 1 DOF, the order of DOF can be chosen arbitrarily. This helps to define twist angles for DOF of the same joints and consequent joints. It is possible to choose the order of DOF, as shown in Figure 8, so that all inboard link twist angles are

ninety degrees except for elbows that have a zero twist angle. This approach simplifies the transformation matrix for the MDH notation, because the transformation matrix uses the cosine and sine functions of the twist angle which are 1 and 0 in this case. Eq.(3.2) is correspondingly simplified in the following equations.

$$T = \begin{bmatrix} \cos\Theta_i & -\sin\Theta_i & 0 & a_{i-1} \\ 0 & 0 & -1 & -d_i \\ \sin\Theta_i & \cos\Theta_i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.3)$$

$$T_{knee} = \begin{bmatrix} \cos\Theta_i & -\sin\Theta_i & 0 & \text{HipLength} \\ 0 & 0 & -1 & 0 \\ \sin\Theta_i & \cos\Theta_i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.4)$$

For elbows, since $\alpha_{i-1} = 0$, and consequently $\sin(\alpha_{i-1}) = 0$ and $\cos(\alpha_{i-1}) = 1$, it follows that ;

$$T = \begin{bmatrix} \cos\Theta_i & -\sin\Theta_i & 0 & a_{i-1} \\ \sin\Theta_i & \cos\Theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.5)$$

$$T_{elbow} = \begin{bmatrix} \cos\Theta_i & -\sin\Theta_i & 0 & \text{upArmLength} \\ \sin\Theta_i & \cos\Theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.6)$$

Using Eq.(3.4) and Eq.(3.6) are more efficient than making four matrix multiplications at run-time. Other segment joints that have more than one DOF may also use Eq.(3.3) or Eq.(3.5) for the same reason. Another simplification occurs when it is recognized that human segments are drawn after applying all DOF transformations of the inboard joints. Instead of multiplying DOF transformation matrices of the same segment

joint at run-time, constructing one transformation matrix for each segment joint in the initialization process eliminates unnecessary matrix multiplications. The transformation matrix given by Eq.(3.7) is an analytic solution for segment joints that have 2 DOF. This solution is defined by multiplication of two transformation matrices that are in the form of Eq.(3.3). The transformation matrix for 3 DOF is also solved and given by Eq.(3.8). Abbreviations in these equations are “1” for “ Θ_1 ”, “2” for “ Θ_2 ”, “3” for “ Θ_3 ”, “c” for cosine function, and “s” for sine function.

The T matrix for two consecutive joints where both inboard link twists are right angles is;

$$T = \begin{bmatrix} c_1*c_2 & -c_1*s_2 & s_1 & c_1*a_1 + s_1*d_2 + a_0 \\ -s_2 & -c_2 & 0 & -d_1 \\ c_2*s_1 & -s_1*s_2 & -c_1 & s_1*a_1 - c_1*d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.7)$$

The T matrix for three consecutive joints where all three inboard link twists are right angles is;

$$T = \begin{bmatrix} c_1c_2c_3 - s_1s_3 & -c_1c_2s_3 + c_3s_1 & c_1s_2 & c_1c_2a_2 + c_1s_2d_3 + c_1a_1 + s_1d_2 + a_0 \\ -c_3s_2 & s_2s_3 & c_2 & -s_2a_2 + c_2d_3 + d_1 \\ c_2c_3s_1 - c_1s_3 & -c_2s_1s_3 - c_1c_3 & s_1s_2 & c_2s_1a_2 + s_1s_2d_3 + s_1a_1 - c_1d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.8)$$

Element multiplication may be simplified by defining common products as $c_{13} = c_1*c_3$, $s_{13} = s_1*s_3$, $cs_{31} = c_3*s_1$, and $cs_{13} = c_1*s_3$.

The expression given by Eq.(3.10) and Eq.(3.11) provide specific segment joint matrices. The transformation matrix of the pelvis in Eq.(3.9) is applied to the whole human figure. It is the world coordinate of the articulated structure as a root segment.

$$T_{\text{pelvis}} = \begin{bmatrix} & & & 0 \\ & \mathbf{R}_{3 \times 3} & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.9)$$

For waist, neck, wrist, and ankle, $a_0 = a_1 = a_2 = 0$ and d_1 is jointName_Y, d_2 is jointName_X, and d_3 is jointName_Z. This,

$$T_{\text{jointName}} = \begin{bmatrix} & & & c_1*s_2*\text{jointName}_Z + s_1*\text{jointName}_X \\ & \mathbf{R}_{3 \times 3} & & c_2*\text{jointName}_Z + \text{jointName}_Y \\ & & & s_1*s_2*\text{jointName}_Z - c_1*\text{jointName}_X \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.10)$$

For shoulder and hip, $a_1 = a_2 = d_1 = 0$, a_0 is jointName_X, d_0 is jointName_Y, and d_2 is jointName_Z. Consequently,

$$T_{\text{jointName}} = \begin{bmatrix} & & & c_1*s_2*\text{jointName}_Z - \text{jointName}_X \\ & \mathbf{R}_{3 \times 3} & & c_2*\text{jointName}_Z + \text{jointName}_Y \\ & & & s_1*s_2*\text{jointName}_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.11)$$

Hardware characteristics also need to be examined to decide whether using these matrices or making run-time matrix multiplication is better. Today, matrix multiplication is provided in hardware, which increases the computation speed of graphics systems. Therefore constructing these matrices in software may be more expensive than making multiplication of three 4x4-transformation matrices. On the other hand, a minimal articulated structure model has 10 segment joints that have 3 DOF. This means that 30 redundant matrix multiplications are made to draw the human figure. If 150 human

figures are simulated in a large scale networked VE, a computer would make 4500 redundant matrix multiplications. If the system used more complex models that have more joints with 3 or 2 DOF, the number of redundant matrix multiplications would increase. If matrix multiplication is implemented in software, 64 multiplication and 48 additions are needed for only two 4x4-matrix multiplications. Reduction for the last redundant row results in 36 multiplies and 27 adds. The number of these computations for 4500 redundant 4x4-matrix multiplications are 162000 multiplies and 121500 adds.

When segment joint transformation matrices are used for the minimal model, only 15 matrix multiplications are calculated at run-time. While this sounds reasonable, a new challenge is the minimizing the computation for construction of segment joint matrices.

C. DISPLACEMENT ELEMENTS

Last column of the homogeneous matrix contains the displacement elements of the transformation. These are translations of the current joint on x, y, and z-axes of the previous joint local coordinates. From Eq.(3.8), the displacement elements are computed as:

$$\text{Translation}_X = (c1*c2*a2) + (c1*s2*d3) + (c1*a1) + (s1*d2) + a0$$

$$\text{Translation}_Y = (-s2*a2) + (c2*d3) + d1$$

$$\text{Translation}_Z = (c2*s1*a2) + (s1*s2*d3) + (s1*a1) - (c1*d2)$$

These computations are needed because transitions are calculated for frame 3 relative to frame 0, where all offsets and lengths are local to previous frames, not to frame 0. Defining segment translation relative to previous segments local coordinates eliminates these calculations. If these transitions are set at the initialization phase, no computation takes place for displacement at run-time. Even though jointName_X, jointName_Y, jointName_Z in Eq.(3.10) and Eq.(3.11) are local to frame 0, the articulated structure

definition treats those constants as they are defined relative to in-between frames. This leads to unnecessary computations. A segment joint transformation matrix can be defined as in Eq.(3.12) whether it has 1, 2 or 3 DOF. The idea behind this is that every joint translates and then rotates on local coordinates of previous joint and by doing this, it defines its own local coordinates for its outboard segment vertices and next outboard segment joints. Thus, in general

$$T = \begin{bmatrix} R_{3x3} & \begin{matrix} \text{Joint_X} \\ \text{Joint_Y} \\ \text{Joint_Z} \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix} \quad (3.12)$$

D. SUMMARY

An articulated structure is a mathematical model that defines posture for human figures. Computation of this model is important for real-time systems, especially when large scale networked VE are constructed that can control and display up to 150 human figures at the same time. Making this structure static avoids all run-time computations that are needed for dynamic reconstruction of the hierarchy. Another issue is the mathematical model of the joints. While MDH notation is the most common method, it results in redundant matrix multiplication for the human body articulated structure. Using the segment joint transformation matrix that is given in Eq.(3.12) reduces the number of matrix multiplications. This approach also incorporates branching joints and links. Displacement elements are also defined more efficiently in the last method. In the first method, numbering and choosing DOF order is a complicated task and may lead to errors, while the second method is easier to understand and uses 15 segment joint coordinates instead of 37 frame coordinates. Rotation parts of the segment joint transformation matrices can be defined by two methods: Euler angles and vector-angle

pair. Quaternions provide another representation of vector-angle pair. The next chapter of this thesis compares quaternion and Euler methods with regard to the efficiency of the system.

IV. COMPARISON OF QUATERNION AND EULER ANGLE MODELS

Transformation of moving objects is usually represented by a homogeneous matrix, which is a 4x4-transformation matrix. The idea behind this is that matrix formulation facilitates easy and efficient manipulation by a computer. On the other hand, all 16 elements of this matrix are not needed to represent a transformation. Last row is trivial, but it is necessary to resize the matrix in square form. This allows calculation of the inverse of a homogeneous matrix. If a given system doesn't use the inverse of this matrix, 3 transition elements and one 3x3-rotation matrix can represent a transformation. The first three elements of the last column include translation information and the embedded 3x3-matrix represents rotation. Both Euler angles and vector-angle pair methods that represent a rotation can be formed in a 3x3-matrix, which is useful for the same reason as using homogeneous matrix forms. Today, many 3D graphic engines, such as OpenGL, use homogeneous matrices for transformations.

Coordinate systems should be defined to apply translations. The coordinate system of the whole scene is called “world coordinates” and any coordinate system that a transformation creates is called the “local” coordinates of that transformation. For all 3D coordinate systems, the user defines 3 axes. The direction and name of these axes is very important for translations. Local axes are the transformations of world axes, so they have the same directions and names relative to the local origin. These definitions are also very important for Euler angle methods. It is possible to name these axes in different ways. In this study, world coordinate axes have the same directions and symbols with OpenGL definition, where x-axis is horizontal and the positive direction is right, y-axis is

perpendicular and positive direction is up, and z-axis goes into screen and positive direction is out. These names and directions are disagree with standard aerospace usage for earth-fixed coordinate systems that takes north, east and down directions as basic reference. This must be remembered whenever dynamic models on earth relative navigation are involved in a simulation study.

A. INTRODUCTION

The most common method that is used to parameterize rotations is to use Euler angles. While it is easy to understand for users, it is inadequate for representing all rotations. Another approach is the vector-angle pair, which eliminates many problems of Euler angles. However, this method isn't efficient for consecutive rotations. Quaternions define the vector-angle pair in another way, which adds new features to the representation. Both methods have pros and cons, which are discussed throughout this chapter.

Euler angles describe rotation as a sequence of rotations about three mutually orthogonal coordinate axes fixed in space. These axes may be world or local coordinates, where rotations act on points in the space. Rotations do not rotate the coordinate axes, which remain fixed. The rotations are applied in a fixed order and subsequent rotations have the effect of rotating in space the axes about which the preceding rotations have been applied [WATT92]. There are 6 possible ways to order 3 sequential rotations on different axes. The precise order in which these rotations are applied lead to different orientations. Instead of fixing axes, these three axes may be embedded in each other like in a Gimbal mechanism. Then, outboard axes rotate inboard axes. This is what happens when MDH notation is used for a joint that has 3 DOF. For human segment motions in

this study, swing and bend rotations are applied on openGL x-axis, twist rotations are applied on openGL y-axis, and bow rotations are applied on openGL z-axis.

A vector-angle pair describes a rotation by a rotation angle about a specific axis. This axis may be defined in any direction, but it passes through the origin of current coordinate system. This notation has 4 elements, 1 angle element (θ) and 3 position elements (x, y, z) for the vector head that defines the rotation axis. The other representation of this method, the quaternion, was discovered by Sir William Hamilton in 1843. Even though quaternions have been around for more than 150 years, the use of the unit quaternion gained popularity in the graphics community only in the mid 1980's. A quaternion is like a complex number with one real and three imaginary parts. While a complex number represents a rotation in two-dimensions, a quaternion represents rotation in three-dimensions. A vector-angle pair is included in its formulation. This notation is hard to understand and use, but it solves many problems of Euler angle methods.

B. VECTOR ROTATION

In this section, 4 types of product are used: scalar multiplication (*), vector dot product (.), vector cross-product (x), and quaternion product (\otimes). Figure 9 represents the basic 2D rotation of a vertex in XY-plane. This rotation is solved analytically by using polar coordinate representation. Eq.(4.4) includes both polar and planar forms of the representation. Eq.(4.5) represents Eq.(4.4) in matrix form.

Euler angles method defines the rotation on three body fixed orthogonal axes by extending 2D rotations to 3D for each axis. Figure 10 shows Euler angles rotations. Eq.(4.6), Eq.(4.7) and Eq.(4.8) represent individual rotation matrices. Matrix multiplication of these matrices produces rotation matrices for multiple rotations. Notice

that different sequences result in different rotation matrices. Eq.(4.9) is an example for rotation in the order of belly, side and nose.

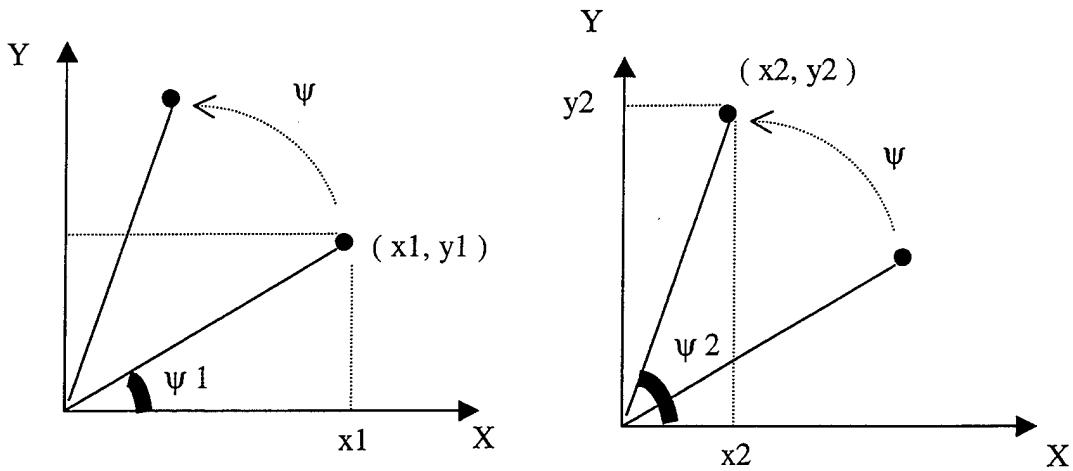


Figure 9:Rotation in 2D

$$\psi 2 = \psi 1 + \psi \quad (4.1)$$

$$r = \sqrt{x1*x1 + y1*y1} = \sqrt{x2*x2 + y2*y2} \quad (4.2)$$

$$(x1, y1) = r * (\cos \psi 1, \sin \psi 1) \quad (4.3)$$

$$\begin{aligned} (x2, y2) &= r * (\cos \psi 2, \sin \psi 2) \\ &= r * (\cos(\psi 1 + \psi), \sin(\psi 1 + \psi)) \\ &= r * (\cos \psi 1 * \cos \psi - \sin \psi 1 * \sin \psi, \cos \psi 1 * \sin \psi + \sin \psi 1 * \cos \psi) \\ &= (x1 * \cos \psi - y1 * \sin \psi, x1 * \sin \psi + y1 * \cos \psi) \end{aligned} \quad (4.4)$$

$$\begin{bmatrix} x2 \\ y2 \end{bmatrix} = \begin{bmatrix} \cos \psi & -\sin \psi \\ \sin \psi & \cos \psi \end{bmatrix} \begin{bmatrix} x1 \\ y1 \end{bmatrix} \quad (4.5)$$

Figure 11 demonstrates the other method, rotation by vector-angle pair. By using vector algebra, Eq.(4.14) is computed. Further solution is possible by replacing p and v with coordinate values. Vector p_1 is the projection of p on v .

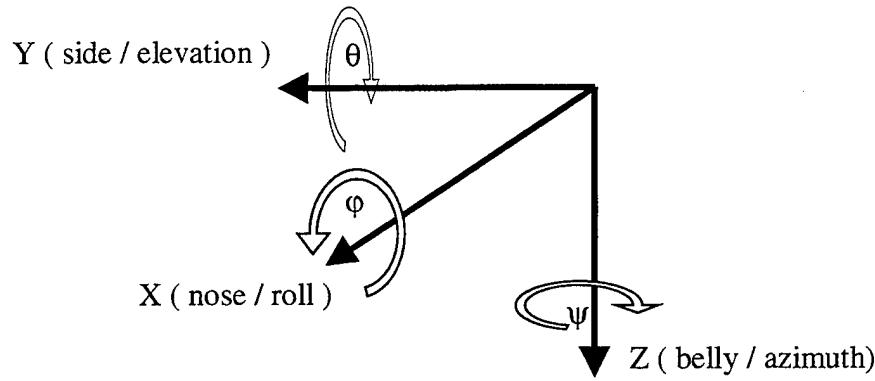


Figure 10: Euler Angles

$$R(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & -s\phi \\ 0 & s\phi & c\phi \end{bmatrix} \quad (4.6)$$

$$R(\theta) = \begin{bmatrix} c\theta & 0 & -s\theta \\ 0 & 1 & 0 \\ s\theta & 0 & c\theta \end{bmatrix} \quad (4.7)$$

$$R(\psi) = \begin{bmatrix} c\psi & -s\psi & 0 \\ s\psi & c\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.8)$$

$$R\psi R\theta R\phi = \begin{bmatrix} c\psi*c\theta & s\phi*s\theta*c\psi - c\phi*s\psi & c\phi*s\theta*c\psi + s\phi*s\psi \\ s\psi*c\theta & s\phi*s\theta*s\psi + c\phi*c\psi & c\phi*s\theta*s\psi - s\phi*c\psi \\ -s\theta & s\phi*c\theta & c\phi*c\theta \end{bmatrix} \quad (4.9)$$

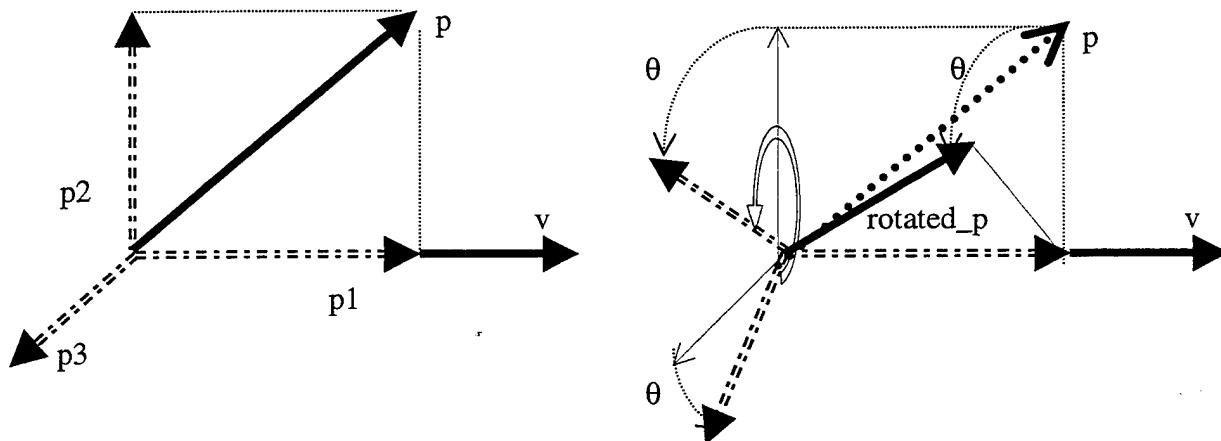


Figure 11: Vector-Angle Pair. (p is rotated on v by θ)

$$p_1 = |p| \cos(\angle p \cdot v) \cdot (v/|v|) = (p \cdot v) v \quad (4.10)$$

$$p_2 = p - p_1 = p - (p \cdot v) v \quad (4.11)$$

$$p_3 = p \times v \quad (4.12)$$

$$p_2_{\text{rotated}} = (\cos \theta) p_2 + (\sin \theta) p_3_{\text{rotated}} \quad (4.13)$$

$$\begin{aligned} p_{\text{rotated}} &= p_1_{\text{rotated}} + p_2_{\text{rotated}} \\ &= (p \cdot v) v + (\cos \theta) (p - (p \cdot v) v) + (\sin \theta) (p \times v) \\ &= (\cos \theta) p + (1 - \cos \theta) (p \cdot v) v + (\sin \theta) (p \times v) \end{aligned} \quad (4.14)$$

The quaternion representation for vector-angle pair is the hardest one. People are used to Euler-angles that make sense to imagine orientation of the object. But with a quaternion, it is impossible to imagine orientation. At the same time, visual demonstration of how the quaternion makes a 3D rotation is also impossible. Actually, a quaternion makes the same rotation as in Figure 11, but with a different approach. To understand quaternions, the representation and algebra of this notation should be covered. Eq.(4.15) gives various representations of quaternions. Eq.(4.16), Eq.(4.17), Eq.(4.18), and Eq.(4.19) introduce the quaternion product for imaginary parts. Notice that it is different than vector cross product. It has a cyclic permutation $i \rightarrow j \rightarrow k \rightarrow i$. Eq.(4.20) and Eq.(4.21) shows scalar product and quaternion addition respectively. The most important operation is the quaternion multiplication Eq.(4.22.), which allows a quaternion to rotate vectors in 3D space. Eq.(4.23.) is the quaternion product for the other representation of quaternion.

Representation :

$$q = (w, v) = (w x y z) = w + x i + y j + z k \quad (4.15)$$

Imaginary part properties (i, j, k have the value $\sqrt{-1}$) :

$$i \otimes i = j \otimes j = k \otimes k = -1 \quad (4.16)$$

$$i \otimes j = k = -j \otimes i \quad (4.17)$$

$$j \otimes k = i = -k \otimes j \quad (4.18)$$

$$k \otimes i = j = -i \otimes k \quad (4.19)$$

Operations :

$$s * q = (s * w, s * v) \quad (4.20)$$

$$q_1 + q_2 = ((w_1 + w_2), (x_1 + x_2), (y_1 + y_2), (z_1 + z_2)) \quad (4.21)$$

$$q_1 \otimes q_2 = (w_1 * w_2 - v_1 * v_2, w_1 * v_2 + w_2 * v_1 + v_1 * x_2) \quad (4.22)$$

$$\begin{aligned} &= (w_1 * w_2 - x_1 * x_2 - y_1 * y_2 - z_1 * z_2) \\ &\quad + (w_1 * x_2 + x_1 * w_2 + y_1 * z_2 - z_1 * y_2) \\ &\quad + (w_1 * y_2 - x_1 * z_2 + y_1 * w_2 + z_1 * x_2) \\ &\quad + (w_1 * z_2 + x_1 * y_2 - y_1 * x_2 + z_1 * w_2) \end{aligned} \quad (4.23)$$

Conjugate :

$$\bar{q} = (w, -v) \quad (4.24)$$

Norm :

$$N(q) = q \otimes \bar{q} = w * w + |v| * |v| = w * w + v * v = |q| * |q| \quad (4.25)$$

Magnitude :

$$M(q) = \sqrt{N(q)} \quad (4.26)$$

Normalized unit quaternion:

$$q / M(q) \quad (4.27)$$

Quaternion inverse:

$$(\text{normal}) \quad q^1 = \bar{q} / N(q) \quad (4.28)$$

$$(\text{Unit quat.}) \quad q^{-1} = \bar{q} \quad (4.29)$$

Eq.(4.29) is also one of the most important properties of the quaternion, which permits the use of Eq.(4.24) to obtain the inverse of a unit quaternion and gives an advantage to the system in computation speed.

Quaternion multiplication of two quaternions results a new quaternion that represents rotation of first quaternion by the second one Eq.(4.30). Eq.(4.22) has a vector cross-product ($v_1 \times v_2$), which means quaternion product is not commutative Eq.(4.31). But it is associative Eq.(4.32). To rotate a vertex by a quaternion, 3 steps are taken. First, vertex is written in quaternion form by adding a 0 scalar part Eq.(4.33). Second, the quaternion is normalized and formed in a unit quaternion Eq.(4.34). Last step is the applying Eq.(4.35). This equation uses inverse of the unit quaternion, which is an efficient computation. Another useful rule is that quaternion multiplication of two unit quaternion always results a single unit quaternion. This is so important for efficient computation to apply rotation consequences. Rotations are expressed in body fixed coordinates.

quaternion (q_1) is rotated by quaternion (q_2) :

$$q_1\text{-rotated} = q_1 \otimes q_2 \quad (4.30)$$

$$q_1 \otimes q_2 \neq q_2 \otimes q_1 \quad (4.31)$$

$$q_1 \otimes (q_2 \otimes q_3) = (q_1 \otimes q_2) \otimes q_3 \quad (4.32)$$

vertex (p) is rotated by quaternion (q) :

$$\text{quat_}p = (0, p) \quad (4.33)$$

$$\text{unit_}q = q / M(q) \quad (4.34)$$

$$\text{rotated_quat_p} = \text{unit_q} \otimes \text{quat_p} \otimes \text{unit_q}^{-1} = (0, \text{rotated_p}) \quad (4.35)$$

vertex (p) is rotated by vector (v) -angle (θ) :

$$q = (\cos(\theta/2), \sin(\theta/2) * v) \quad (4.36)$$

Eq.(4.36) gives the representation of a vector-angle pair by a quaternion. This helps to imagine quaternion rotation. Because, it is possible to extract vector-angle pair from quaternion definition or to input quaternion to the system as a vector-angle pair. If analytic solution is made for Eq.(4.35) by using Eq.(4.36) and Eq.(4.22), result is the same with Eq.(4.14) for rotated_p, Eq.(4.37).

$$\begin{aligned} \text{rotated_quat_p} &= (0, (\cos\theta) p + (1-\cos\theta) (p.v) v + (\sin\theta)(p \times v)) \\ &= (0, \text{rotated_p}) \end{aligned} \quad (4.37)$$

In composing successive quaternions to obtain the resultant total rotation quaternion, it is important to remember that rotation about body-fixed axes multiply on the right while rotations about earth fixed axes multiply on the left, of course, the same is also true of matrix representation of rotations.

C. CONVERSION TO HOMOGENEOUS MATRIX

Next step is to define homogeneous matrices with these methods. Eq.(4.9) gives the rotation matrix of 3 Euler angles. Eq.(4.38) is constructed by adding displacement elements to this matrix. As mentioned before, a redundant last row is also added. Simplifications are made to gain computation speed ($c\varphi\psi = c\varphi*c\psi$, $s\varphi\psi = s\varphi*s\psi$, $cs\psi\varphi = c\psi*s\varphi$, and $cs\varphi\psi = c\varphi*s\psi$).

$$R\psi R\theta R\varphi = \begin{bmatrix} c\psi*c\theta & cs\psi\varphi * s\theta - cs\varphi\psi & c\varphi\psi * s\theta + s\varphi\psi & X \\ s\psi*c\theta & s\varphi\psi * s\theta + c\varphi\psi & cs\varphi\psi * s\theta - cs\psi\varphi & Y \\ -s\theta & s\varphi * c\theta & c\varphi * c\theta & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.38)$$

The second method, vector-angle pair, is given by Eq.(4.14). Further analytical solution of this equation produces the matrix form of this rotation. Eq.(4.40) represents the simplified matrix form for vector-angle pair rotation. Eq.(4.39) is used in OpenGL for transforming objects.

$$R(v(x,y,z), \theta) = \begin{bmatrix} x*x*(1-c\theta) + c\theta & y*x*(1-c\theta) + z*s\theta & x*z*(1-c\theta) + y*s\theta & X \\ y*x*(1-c\theta) + z*s\theta & y*y*(1-c\theta) + c\theta & y*z*(1-c\theta) + x*s\theta & Y \\ z*x*(1-c\theta) + y*s\theta & y*z*(1-c\theta) + x*s\theta & z*z*(1-c\theta) + c\theta & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.39)$$

Simplifications are made as $ci = 1 - c\theta$, $xyci = y*x*ci$, $xzci = z*x*ci$, $zyci = y*z*ci$, $xs = x*s\theta$, $ys = y*s\theta$, and $zs = z*s\theta$. The result is:

$$R(v(x,y,z), \theta) = \begin{bmatrix} x*x*ci + c\theta & xyci + zs & xzci + ys & X \\ xyci + zs & y*y*ci + c\theta & zy*ci + xs & Y \\ xzci + ys & zyci + xs & z*z*ci + c\theta & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.40)$$

Eq.(4.35) gives quaternion rotation for a vertex. Analytic solution of this equation can be done by using Eq.(4.23). This solution results the 3x3-matrix form of the rotation embedded in Eq.(4.42).

$$R = [q \otimes i \otimes q^{-1} \quad | \quad q \otimes j \otimes q^{-1} \quad | \quad q \otimes k \otimes q^{-1}] \quad (4.41)$$

or

$$R(w, v(x,y,z)) = \begin{bmatrix} 1 - 2*y*y - 2*z*z & 2*x*y - 2*w*z & 2*x*z + 2*w*y & X \\ 2*x*y + 2*w*z & 1 - 2*x*x - 2*z*z & 2*y*z + 2*w*x & Y \\ 2*x*z - 2*w*y & 2*y*z - 2*w*x & 1 - 2*x*x - 2*y*y & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.42)$$

Simplifications are made as $xx = x*x$, $zz = z*z$, $yy = y*y$, $xy = x*y$, $zy = z*y$, $xz = x*z$, $wx = w*x$, $wx = w*x$, and $wx = w*x$, resulting in:

$$R(w, v(x,y,z)) = \begin{bmatrix} 1 - 2*(yy+zz) & 2*(xy - wz) & 2*(xz + wy) & X \\ 2*(xy + wz) & 1 - 2*(xx+zz) & 2*(yz - wx) & Y \\ 2*(xz - wy) & 2*(yz - wx) & 1 - 2*(xx+yy) & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.43)$$

Even though all three notations may use other formulations instead of using homogeneous matrix form, computation efficiency for construction of segment joint transformation matrices are discussed in this section because of the reasons reviewed in Chapter III. Figure 12 shows a graph to compare three notations for rotating a segment joint that has a 3 DOF. This rotation also represents the orientation, because it is relative to the origin of the local coordinate system. Notice that quaternion notation doesn't use trigonometric functions, which is an advantage for computation speed.

The quaternion normalization process also adds extra computation. But as mentioned before successive quaternion products of unit quaternions results in a unit quaternion, which means that normalization is not needed in each step, but only periodically, to correct for accumulated round off effects,

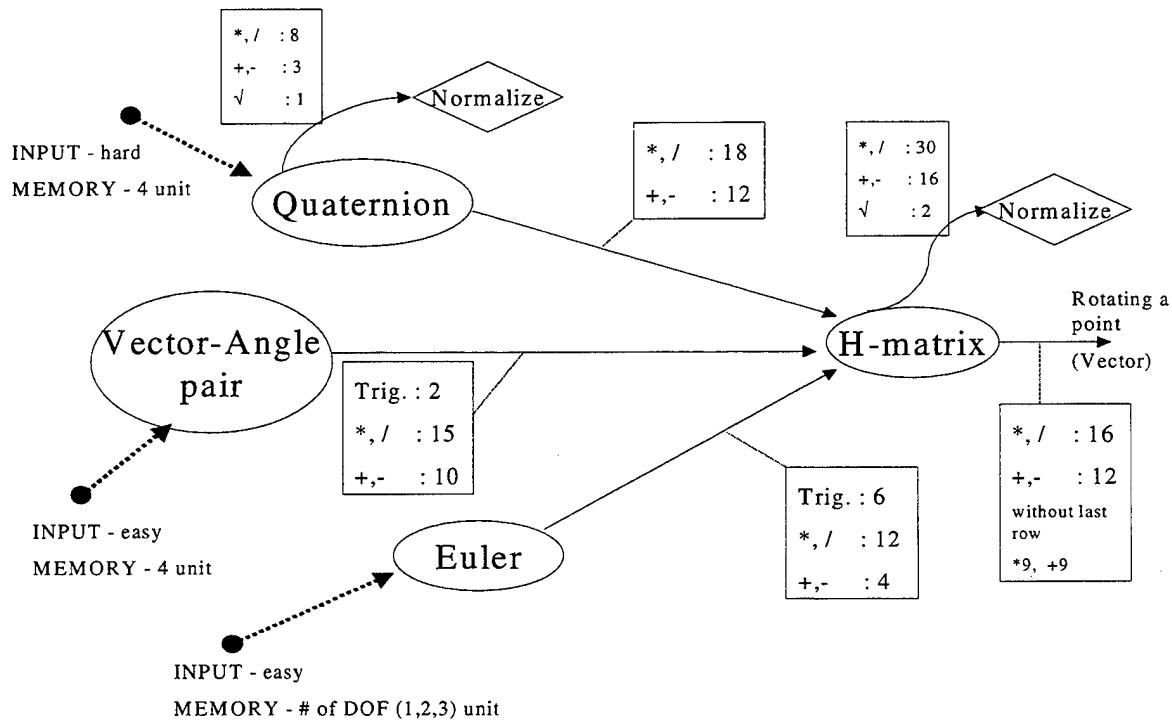


Figure 12: Comparison of Methods for an Orientation

Another important comparison of notations is for rotations on existing orientations. Existing orientations and applied rotations are used to compute the new orientations. A pair of Euler-angles that each represents 3 DOF can be added to compute total rotation, but there are many combinations of these additions. Matrix multiplication of two homogeneous matrices may be a solution for this. Rotations of two vector-angle pair can be merged by using two homogeneous matrices, or they can be converted quaternion to create single unit quaternion. The last approach is more efficient [Figure 13].

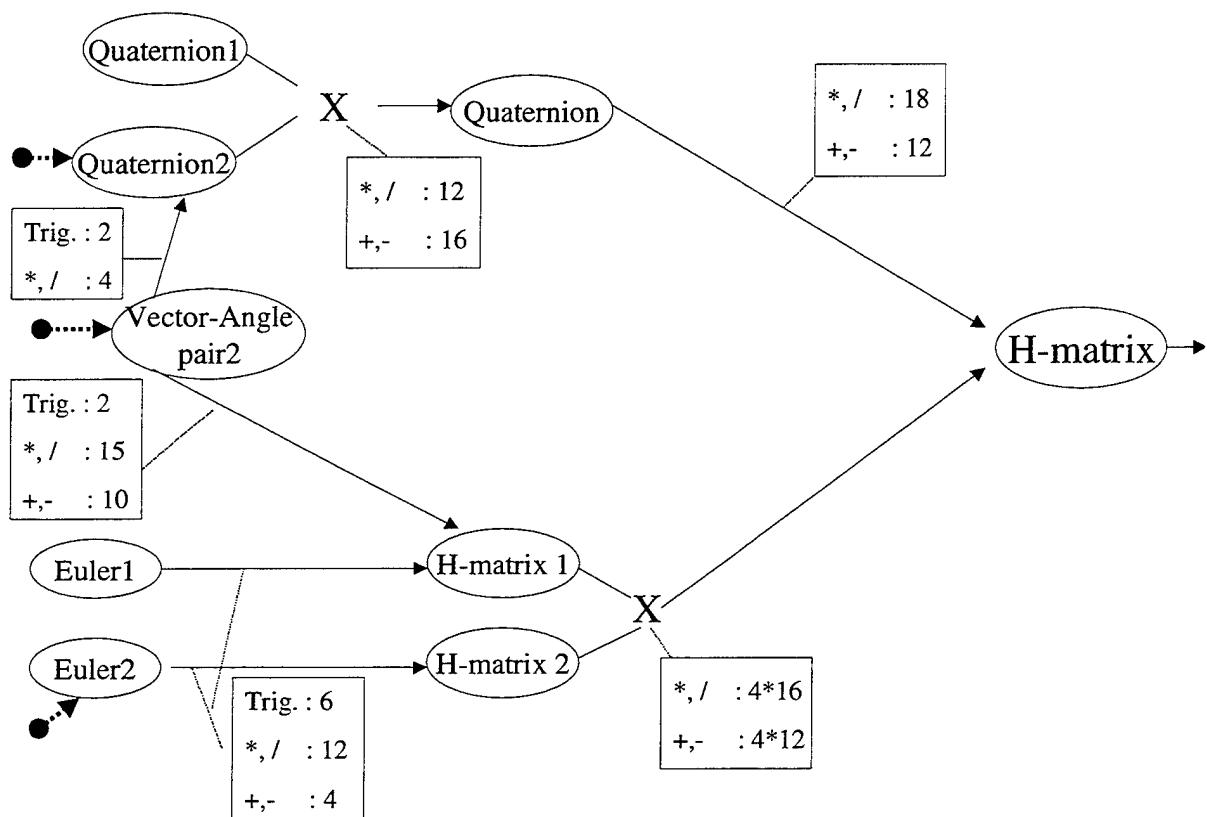


Figure 13: Comparison of Methods for a Rotation from an Existing Orientation

D. INTERPOLATION AND SINGULARITY

A software system may interpolate between two orientations to fill the gaps in motion. Interpolation is very important for key frame systems. Motion should be smooth in the desired direction and speed. For rotating objects, all three notations (Euler angles, vector-angle pair and quaternion) may be used to define the key frames.

If Euler angles used for joints that have more than 1 DOF, motion between two key frames will be undefined unless some constraints on Euler angle rates are applied, such as, for example linear interpolation of all angles. Even in such a case, however, a change in the order of rotation axis will produce a different motion. Euler angles are therefore undesirable for interpolation and should be converted other forms for interpolation purposes. This adds extra computation at run time.

Figure 14 shows the singularity of Euler angles. When the elevation angle reaches 90 or -90 degrees, 1 DOF is lost. Specifically, roll and azimuth have the same effect. In this case, only two angles are needed to specify the orientation of a rigid body. For example, elevation angle (pitch) and the azimuth (heading) of the belly vector of the body are one such set of angles.

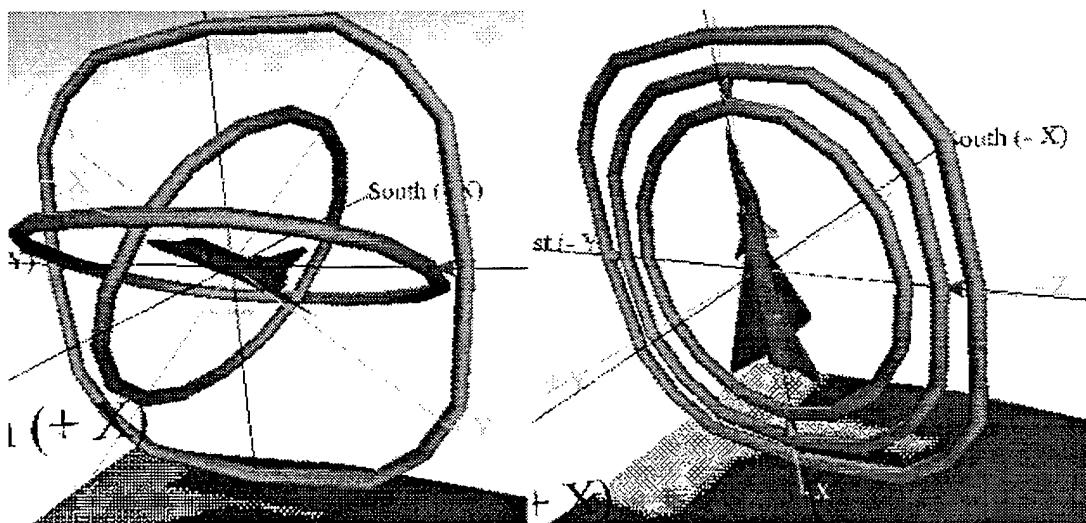


Figure 14: Gimbal Lock (Airplane is attached to innermost ring) [WEBREF3]

Two vectors can represent two orientations of a rigid body. These two vectors lie in a single plane in 3D. Rotation about the normal vector of this plane guarantees a smooth rotation independently of the choice of the coordinate systems or Euler angles. If vector-angle pairs define orientations, calculating the normal vector and rotation angle requires computation time. On the other hand, there is an efficient way to interpolate between two quaternions. Two vector-angle pairs can use this method after being converted to two unit quaternions. Spherical interpolation between two unit quaternions, q_1 and q_2 is given by [WATT92]. The parameter u in Eq.(4.45) is increased from 0 to 1 during interpolation.

$$q_1 \cdot q_2 = \cos \Omega \quad (4.44)$$

$$q(u) = q_1 * \sin((1-u)*\Omega) / \sin \Omega + q_2 * \sin(\Omega*u) / \sin \Omega \quad (4.45)$$

In the above, a problem occurs for the direction of interpolation. Every 3D rotation has two representations in quaternion space. The effect of q and $-q$ are the same. This means that interpolation can be computed using either quaternion, which causes two motions in opposite directions during interpolation. To solve this ambiguity, interpolations should occur either between quaternion pairs q_1 and q_2 or q_1 and $-q_2$ for the shortest path [WATT92]. Another potential problem arises as Ω is 0 or π since in this case $\sin \Omega$ is 0. However, this problem is not real, since in this case, interpolation is not needed.

Another advantage of the quaternion method comes up when the system uses body rates. An example is rigid body dynamics. A state chart for such a computation is shown in Figure 15. This system computes linear and angular velocities in earth coordinates by using linear and angular accelerations in body coordinates. Eq.(4.46) represents the matrix to calculate angular rates in earth coordinates for Euler angle methods. This

equation can be understood by observing Figure 14. Analytic solution of this matrix results Eq.(4.47). The secant function introduces the singularity problem of this approach, when θ is 90 degrees.

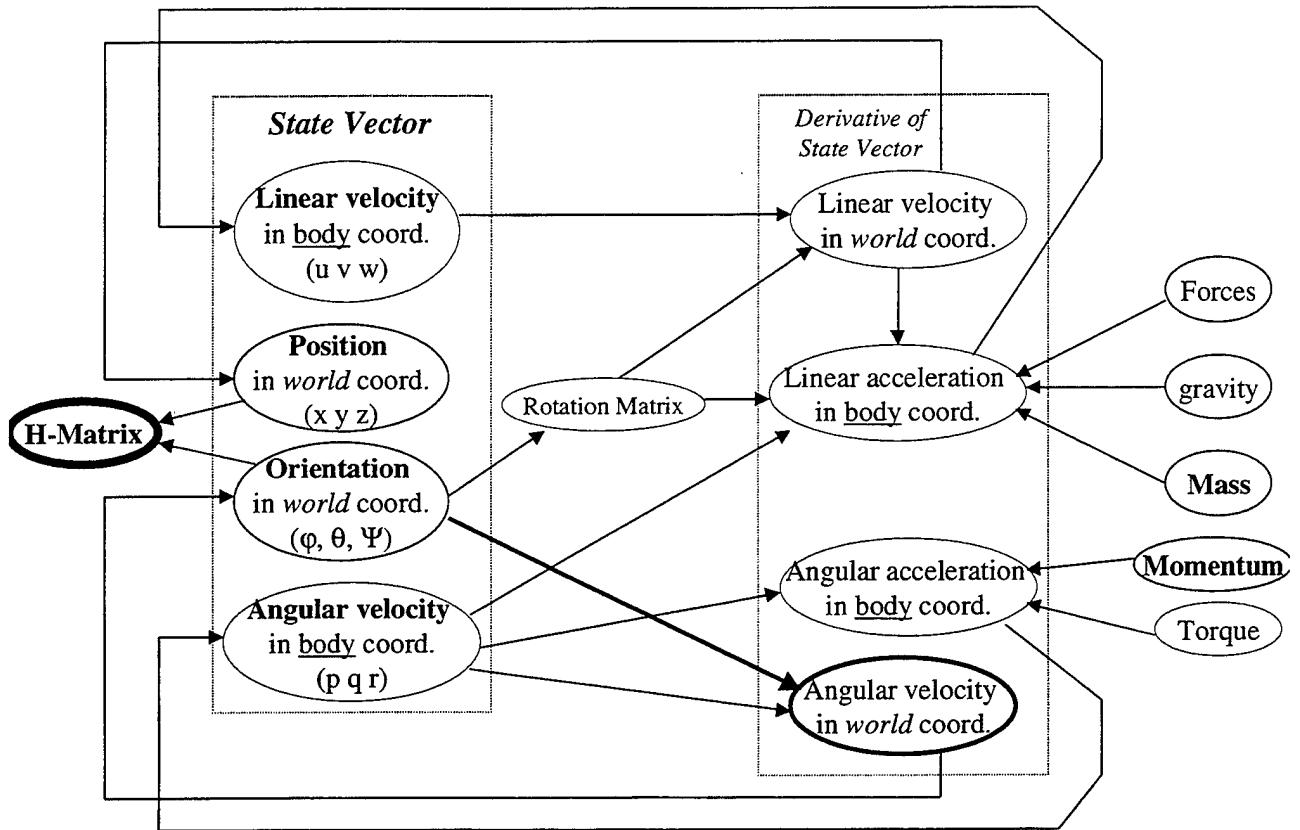


Figure 15: Loop for Rigid Body Dynamics

$\mathbf{Ev}(\phi \theta \psi)$: Angular velocity in earth coordinate

$\mathbf{Bv}(p q r)$: Angular velocity in body coordinate

$$\mathbf{Ev} = \mathbf{R}\psi \mathbf{R}\theta \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + \mathbf{R}\psi \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} = \mathbf{R}\psi \mathbf{R}\theta \mathbf{R}\phi \mathbf{Bv} \quad (4.46)$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \tan\theta * \sin\phi & \tan\theta * \cos\phi \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sec\theta * \sin\phi & \sec\theta * \cos\phi \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (4.47)$$

On the other hand, as shown by Eq.(4.48) below, the quaternion method shows no singularity to convert body rates to earth coordinates [COOK92].

$$q\text{-earth} = q \otimes (0 \ p \ q \ r) / 2 \quad (4.48)$$

E. CONSTRAINT DEFINITION

Computer simulation of articulated bodies may involve constraints on motion. One important type of constraint is the joint angle constraint. For example, an animator can work easier after defining joint angle constraints, so impossible motions are disregarded or corrected by the simulation system. Another example is motion-tracking systems, where system software checks sensor inputs for validity. Defining joint constraints by quaternions is almost impossible, because a quaternion is a 4D vector and doesn't have a graphical representation. Vector-angle pairs may be used for constraint definition. But a more intuitive way is to use Euler angle methods. Table 1-5 in Chapter III gives constraints for each joint DOF. They are reasonable and understandable. But these constraints, as given are not realistic for human motion. This is because joint motion limits are not independent. For example for the human shoulder joint, the maximum values for roll and elevation depend on each other and also upon shoulder azimuth.

In addition to such considerations, constraints should also prevent segment collisions. If a system uses quaternions to represent joint orientations, conversion to Euler angles should be done to set constraints at run-time. This can be accomplished by

computing an equivalent rotation matrix for the quaternion, and then solving the inverse kinematics problem for the matrix to obtain Euler angles.

F. HARDWARE, SOFTWARE AND NETWORK CONSIDERATIONS

Existing hardware implementations and 3D graphic APIs support matrix forms and multiplication, as an important advantage for real-time systems. This is why the homogeneous matrix computation is useful and most common. On the other hand, the quaternion method may be used without converted to homogeneous matrix notation. This approach uses some advantages of unit quaternion methods. A quaternion can be normalized with a minimal amount of computational expense, whereas normalization of a rotational matrix requires substantially more effort. In a situation where continuous chains of transformational products are relatively rare and highly parallel hardware is available, homogeneous matrices may prove superior to quaternion/vector pairs. Conversely, if the nature of a computation necessitates frequent renormalization of rotational operators, then the quaternion/vector approach arises as the better alternative [FUND90]. Notice that Euler angles have to be converted to homogeneous matrix form to be normalized.

Inserting human models into networked synthetic environments necessitates the exchange of posture information between network nodes. Posture information in a protocol data unit (PDU) includes all DOF of an articulated rigid body. Another approach is sending only the updated DOF. But the last method needs extra computation for packing and unpacking of PDUs, which makes the first method better. The number of bytes representing articulation of a minimal model is 34 when Euler angles method is used. This number becomes 56 when quaternions are used. The difference is 22 bytes and increases when complex articulated bodies are used. The quaternion method adds 1 byte

for 3 DOF, 2 bytes for 2 DOF and 3 bytes for 1 DOF to the number of bytes needed by Euler angle methods. When more than 150 humans are inserted to a simulation, the quaternion representation requires 3300 extra bytes for minimal articulated bodies.

G. USER INTERACTION

When low level-motion control systems are used, user interaction techniques for all three notations should be defined. Input boxes or a script console may be used for vector-angle and Euler methods. But quaternion numbers don't make sense to an animator.

Using a mouse as an interaction device and choosing 2D graphical interaction is the most common way used today for human figure control. Objects may be embedded in their local coordinate circles to make visual perception much stronger. Figure 16 shows an intuitive way for vector angle pairs.

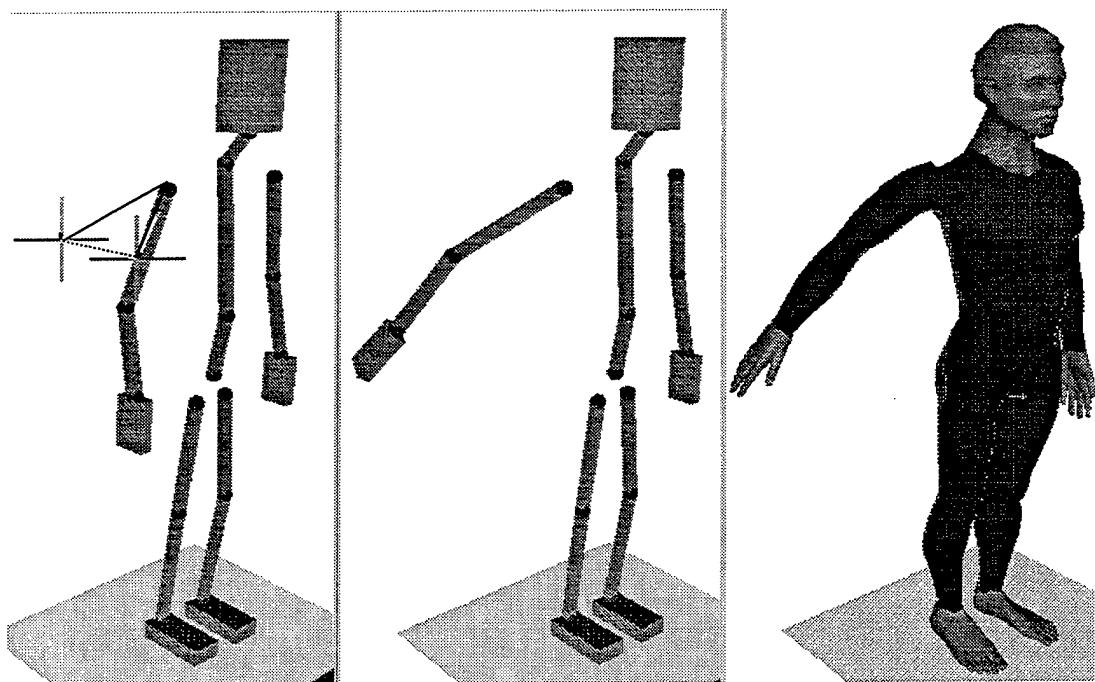


Figure 16: A 2D Graphical Interaction Method for Vector-Angle Pair

In this system, mouse inputs are used to rotate the whole world when the mouse is moved, and the same motion defines a 2D angle when mouse button is first clicked on a segment. Angles are calculated by converting of mouse cursor position from screen coordinates to world coordinates. This is achieved by reversing the projection process. An angle is measured on a plane that is parallel to the screen and passes through a joint origin. The vector is perpendicular to the screen and the direction is out. This method may also be used for quaternion notation after a vector-angle pair is converted to a unit quaternion. Actually this interaction lets the user experience the advantages of unit quaternions.

Another method for low-level inputs is to use motion-tracking systems. Miniature angular rate sensors may be used to measure angular rates of individual human segments in body coordinates with drift connection from accelerometers and magnetometers. When a system uses these inputs to calculate posture information in earth coordinates, Euler angles method fails for 90 degrees rotation on second axes. Using quaternions overcome this singularity [DUMA99].

H. SUMMARY

All three notations have cons and pros. A system may use a combination of all for optimization purposes. From a computation perspective, quaternions seem to have some advantages. But these advantages are not so decisive and may be lost when hardware implementations for matrix notation are considered. Look-up tables may be added to a software system for trigonometric functions. The quaternion method eliminates the singularity problems of the Euler angle method. The Euler angle method is inadequate for interpolation, but it is better for construction of PDUs and to set constraints for joints.

V. FORWARD AND INVERSE KINEMATICS

Kinematics models were developed and used by researchers in robotics long before computer graphics came along, and so there is a wide body of literature devoted to them [CRAI89]. All kinematics models specify motion independent of the underlying forces and include all geometrical and time related properties of motion, such as position, velocity and acceleration. Kinematics animation of articulated structures usually falls into one of two categories: forward and inverse kinematics. Both have been used in computer animation.

A. DEFINITION

In forward kinematics, all transformations are specified to control motion of an end effector. This is simple and has low computation requirements. The importance of interactive real-time design in animation makes forward kinematics systems attractive. In this approach, the animator has direct control over the figure's position. All issues that are introduced in Chapter III and Chapter IV of this thesis are related to forward kinematics. This chapter introduces inverse kinematics and compares both methods.

In inverse kinematics, a goal position and orientation is specified for the end effector and the system computes the transformations required to achieve this goal. This is also called “goal directed motion” since the animator only specifies the end effector's position and orientation. The system software then solves for set of joint angles that place the end effector in the desired posture. This method is generally very expensive with respect to computational requirements. However, the importance of automatic motion control makes inverse kinematics systems attractive for some applications, where movement of an end effector drives the animation. In such cases, forward kinematics solutions are

counterintuitive and tedious. Inverse kinematics can be used where precise motion control is not required, as with, for example, autonomous agents in a networked virtual training and simulation environment.

Hybrid models may be constructed by using both methods. For example, arms and legs can be controlled by inverse kinematics, while the torso and neck are simulated by forward kinematics. Inverse kinematics algorithms can compute orientations of the shoulder, elbow and wrist to reach an object in space. Leg motions are also computed in the same way to control the interaction between toes and the ground. Another example is a human figure sitting on a bicycle. The end nodes of the hands are constrained to be on the handles and those of the feet to the pedals. As the pedals are made to revolve, or as the handle bar turns, then the feet and hands follow accordingly [WATT92].

B. COMPUTATION

Both methods become harder to use as the complexity of the articulation increases. Complexity can increase due to an increase in the number of DOF and number of links. Chapter III introduces computational issues of forward kinematics. While forward kinematics should check joint constraints and collisions, inverse kinematics should additionally check reachable space.

Inverse kinematics solutions are grouped into two broad classes: closed form solutions and numerical solutions. Numerical solution is a whole field of study itself and is beyond the research of this thesis, but some computational issues of this method are discussed below. A numerical solution is also called an iterative solution. This method uses the relation between end effector velocities and state variable rates. In generally, if X is end effector state vector (position and orientation) and θ is system state vector

(orientations of in-between links), there is a function f of θ to compute X , which is given as: $X = f(\theta)$. The derivative of both sides of this equation gives a relation between velocities ($dX = J(\theta)d\theta$). The matrix J is called the Jacobian, which maps velocities in state space to velocities in cartesian space. The general inverse kinematics problem is given as: $\theta = f^{-1}(X)$. But the function f is highly nonlinear, rapidly becoming more and more complex as the number of links increases, and so the inversion of this function soon becomes impossible to perform analytically. The problem can be made linear, however, by localizing about the current operating position and inverting the Jacobian to give: $d\theta = J^{-1} dX$ [WATT92]. The matrix J can be constructed by the information extracted from transformation matrices that already exist in the graphics pipeline. The first problem is that J is not square in many cases. In such cases pseudo inversion techniques can be used to compute J^{-1} . Iterating end effector state variables, described as dX , is the second problem, which needs extra computations to correct tracking errors. Singularity problems and joint constraints are other issues to be solved when an end effector is iterated. Because of these problems, numerical solutions are much slower than the corresponding closed form solution and not preferred in many cases. They are not always real-time and lead to only one solution even though there is more than one solution to reach the goal in many cases.

Closed form solutions are based on analytic expression and can be hard coded, which is real-time and more preferable. This method is also grouped into two sub classes: algebraic and geometric solutions. The two methods differ perhaps in approach only [CRAI89]. The problem of closed form solutions is that when the number of joints increases, the inverse kinematics problem is generally not solvable. When a manipulator

has less than six degrees of freedom, it cannot attain general goal positions and orientations in 3D space. In many realistic situations, manipulators with four and five degrees of freedom are employed which operate out of a plane, but clearly cannot reach general goals. A sufficient condition that a manipulator with six revolute joints will have a closed form solution is that three neighboring joint axes intersect at a point [CRAI89]. Another problem is that the more nonzero link parameters there are, the more ways there will be to reach a certain goal. Only one solution should be chosen at run time, which necessitates extra algorithms and computations.

Figure 17(a) shows a human arm with 6 DOF, where the shoulder has 2 DOF, the elbow has one DOF and the wrist has 3 DOF. The third segment translation, d_2 , is on the fourth axis. Thus, the translation d_2 and rotation θ_4 has the same effect as rotation θ_4 is applied before d_2 displacement. This allows the alternative of a 2 DOF elbow. Figure 17(b) shows this change. Eq.(5.1-6) give transformation matrices for Figure 17(b).

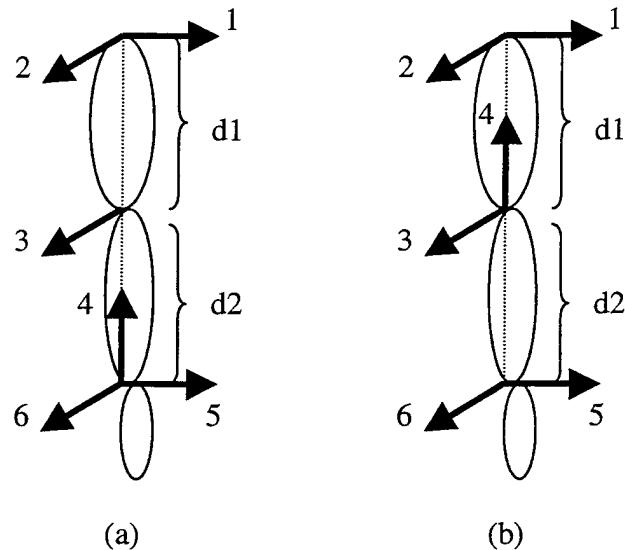


Figure 17: Human Arm

The position and orientation of a hand is defined by matrix multiplication in the order of R1, R2, T1, R3, T2, R4, R5, and R6. This multiplication can be re-ordered as R1, R2, T1, R3, R4, T2, R5, and R6. Consequent homogeneous matrices are defined as H1, H2, H3 H4, H5, and H6 as follows:

$$H1 = R1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c1 & -s1 & 0 \\ 0 & s1 & c1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.1)$$

$$H2 = R2 = \begin{bmatrix} c2 & -s2 & 0 & 0 \\ s2 & c2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.2)$$

$$H3 = T1 R3 = \begin{bmatrix} c3 & -s3 & 0 & 0 \\ s3 & c3 & 0 & d1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.3)$$

$$H4 = R4 = \begin{bmatrix} c4 & 0 & s4 & 0 \\ 0 & 1 & 0 & 0 \\ -s4 & 0 & c4 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.4)$$

$$H5 = T2 R5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c5 & -s5 & d2 \\ 0 & s5 & c5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.5)$$

$$H6 = R6 = \begin{bmatrix} c6 & -s6 & 0 & 0 \\ s6 & c6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.6)$$

Notice that 1 DOF (twist) has been omitted at the shoulder, which may be important for some applications that necessitate accurate representation of human motions. Since, axis 4,5 and 6 are intersecting, there is a solution for this system that has 6 DOF. Eq.(5.9) and Eq.(5.13) can be computed by using Eq.(5.7) and Eq.(5.8). The approach in this

method is the inverting homogeneous matrices in specific order, bringing to the left side and solving for the equation where there is only one unknown. If no more equations can be found, invert next matrix and solve further. Eq.(5.17) is found after second step Eq.(5.14). Last step is given by Eq.(5.18-19), which allows the solution of angle 4 through angle 6 Eq.(5.21-27).

$$H = T R \Rightarrow H = R T = R T \quad (5.7)$$

$$H_1^{-1} H_{hand} = H_2 H_3 H_4 H_5 H_6 \quad (5.8)$$

$$\text{Angle 1} = \arctan(Dz/Dy) \quad (5.9)$$

$$a = -2*d1*(c1*Dy + s1*Dz) \quad (5.10)$$

$$b = 2*Dx*d1 \quad (5.11)$$

$$c = d2^2 - d1^2 - Dx^2 - (c1*Dy + s1*Dz)^2 \quad (5.12)$$

$$\text{Angle 2} = \arctan(b/a) \pm \arctan(\sqrt{a^2 + b^2 - c^2} / c) \quad (5.13)$$

$$H_2^{-1} H_1^{-1} H_{hand} = H_3 H_4 H_5 H_6 \quad (5.14)$$

$$s3 = (c2*Dx + s2*c1*Dy + s2*s1*Dz) / -d2 \quad (5.15)$$

$$c3 = (-s2*Dx + c2*c1*Dy + c2*s1*Dz - d1) / d2 \quad (5.16)$$

$$\text{Angle 3} = \arctan(s3 / c3) \quad (5.17)$$

$$H_4^{-1} H_3^{-1} H_2^{-1} H_1^{-1} H_{hand} = H_5 H_6 \quad (5.18)$$

$$(R3 R4)^{-1} T1^{-1} (H1 H2)^{-1} H_{hand} = H_5 H_6 \quad (5.19)$$

$$0 = c4*(Dx*c23 + s23*((c1*Dy) + (s1*Dz)) - d1*s3) - s4*(-s1*Dy + c1*Dz) \quad (5.20)$$

$$\text{Angle 4} = \arctan((Dx*c23 + Dy*s23*(c1+s1) - d1*s3) / (-s1*Dy + c1*Dz)) \quad (5.21)$$

$$c5 = s4*(Ax*c23 + s23*((c1*Ay) + (s1*Az)) - d1*s3) + c4*(-s1*Ay + c1*Az) \quad (5.22)$$

$$s5 = s3*(c2*Ax + c1*s2*Ay + s1*s2*Az) - c3*(-s2*Ax + c1*c2*Ay + s1*c2*Az - d1) \quad (5.23)$$

$$\text{Angle 5} = \arctan(s6 / c6) \quad (5.24)$$

$$c6 = c4 * (Nx * c23 + s23 * ((c1 * Ny) + (s1 * Nz)) - d1 * s3) - s4 * (-s1 * Ny + c1 * Nz) \quad (5.25)$$

$$s6 = -c4 * (Ox * c23 + s23 * ((c1 * Oy) + (s1 * Oz)) - d1 * s3) + s4 * (-s1 * Oy + c1 * Oz) \quad (5.26)$$

$$\text{Angle 6} = \arctan(s6 / c6) \quad (5.27)$$

This solution can be applied to both arms. A similar solution can be found for the legs, where the twist motion of hips has been omitted in this model. The solution of all these equations are specified for the given axis order. If the drawing system uses a different axis order, extra computations are needed to switch between different axis systems. This is because the solutions above are based on Euler angles. There isn't a similar study for algebraic solution of quaternions. [FUND90] uses quaternion/vector pair for a formalism to solve the inverse kinematics problem for a six-jointed revolute manipulator with a spherical wrist. This is a quaternion-based solution to compute Euler angles.

C. SENSOR PLACEMENT

One of the important application areas that use human figure models is motion tracking systems. These systems may use both forward and inverse kinematics. Actually, choosing one or the other is a trade-off for realism. Figure 18 shows an inverse kinematics approach. Four sensors are used to track the human upper body. The advantage of this approach is that user encumbrance is reduced by using a minimal number of sensors. The disadvantage is that more computations are needed. Also, because of multiple solutions, the result may be inaccurate. The position of end-effectors, the hand's position in this case, must be tracked as well as orientations, which is another disadvantage.

A second approach is introduced in Figure 19. Using 15 sensors, it is possible track each limb segment by using forward kinematics. The system tracks only one segment's position and needs only the orientations of all other segments. [SKOP96] demonstrates the tracking of an arm in real-time by using forward kinematics. [SKOP96] also uses inverse kinematics solutions to determine joint angles associated with physical limb on which the tracker is mounted. This is because of his use of solutions based on Euler angles. Another approach is to use quaternions. [DUMA99] explains a quaternion filter that computes a unit quaternion from sensor data. When quaternions are used for a forward kinematics approach, there is no need for inverse kinematics computations.

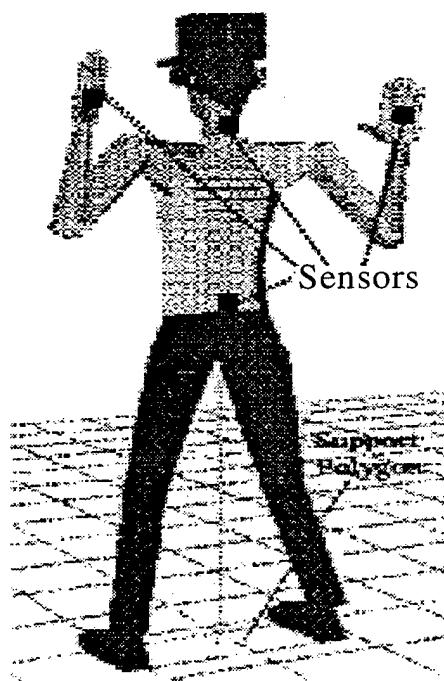


Figure 18: A Minimally Sensed Human
[SKOP96]

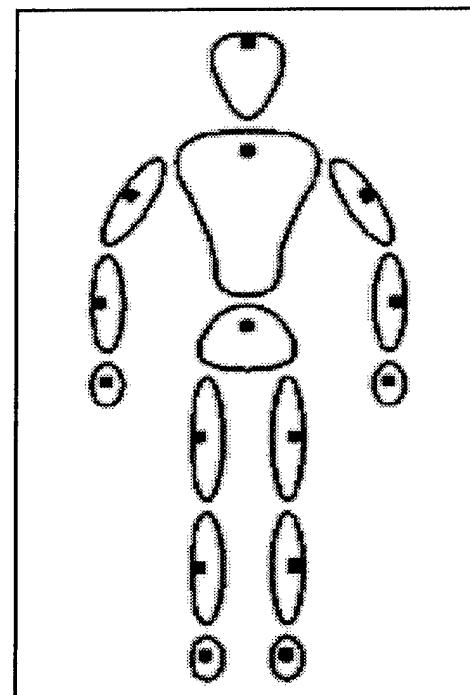


Figure 19: Proposed Hybrid Human
Tracking Sensor Config. [FREY96]

D. SUMMARY

Both forward and inverse kinematics have application areas in computer graphics. It is also possible to model a human articulated rigid body system with inverse kinematics. But the main concern is that some DOF may need to be omitted for real-time animation. Inverse kinematics solutions will also lead to multiple postures and system software will choose one of them, which may be inaccurate in special cases. Iterative solutions can be used when human models are used in non real-time applications. Another problem with the iterative approach is that the end effector path must be well defined. In many cases, an animator needs to switch to forward kinematics to correct inverse kinematics solutions.

[FREY96] shows that an entire human body can be tracked by using only orientation data for each body part. This result eliminates the need for human body motion capture systems to track the position of each body part as well as the need to create highly complex kinematics models of the human body based on joint angles. Using unit quaternion data simplifies the system further and permits the construction of an articulated body that doesn't use inverse kinematics computations [DUMA99].

VI. IMPLEMENTATION AND RESULTS

The simulation program of this thesis is developed by using OpenGL and GLUT libraries and allows a user to define human figure postures by means of mouse inputs. Further, in this thesis, a key frame animation system is also developed to interpolate between user-defined frames. Both forward kinematics algorithms in [WATT92] and inverse kinematics algorithms in [BEDI97] are used to demonstrate walking of the figure. The main purpose of the program in Appendix B is to build the human articulated structure with quaternions. Approaches to define user interaction and joint constraint definition for quaternions are represented in detail in following paragraphs of this chapter.

There are currently 20 classes in the program (Figure 20). The main class is *GlutBaseClass* uses GLUT library and constructs a window to draw the human figure and to collect user inputs. Sub-main classes are Human, UserControl, ProceduralAnim and KeyFrameAnim classes. These classes are static and instantiated once. Three types of control of the figure are implemented in the last three sub main classes that have a pointer to a human object for direct manipulation. The *KeyFrameAnim* object stores and manages the orientation key values of segments in Posture objects as quaternions, and interpolates between currently stored frames for animation. The *ProceduralAnim* object controls the human figure for walking. The *UserControl* object tracks mouse motion and creates manipulation of segments, depending on user choice of control type. Control types are forward kinematics with Euler angles, forward kinematics with quaternions and simple algebraic inverse kinematics. The UserControl object feeds back user inputs by animation of a GimbalSystem object for Euler angles and by animation of Cursor3D

objects for quaternions and inverse kinematics. Menus that are represented by *GlutBaseClass* allow choices for control type, scene navigation type, and segment shape. The *GlutBaseClass* also animates a floor object trivially to demonstrate walking.

Another control type uses sensor inputs. [DUMA99] demonstrates a quaternion filter, which produces a unit quaternion by using three types of sensors. This system can track a human segment in real-time. His *Qaef* object is embedded into a *SensorSystem* object, which tracks the upper-arm of the human figure.

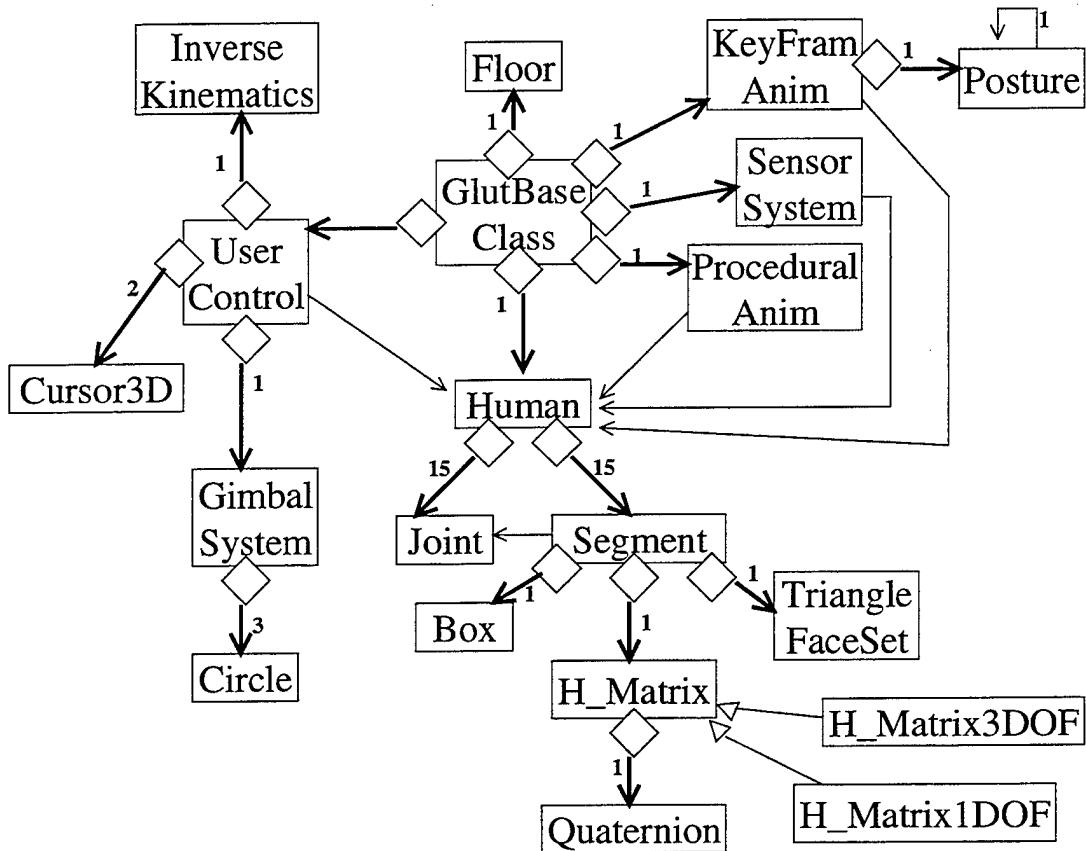


Figure 20: Object Diagram of the Human Program

A. HIERARCHY

The **Human** object has the interface for manipulation of human figure. Hierarchy details and algorithms are embedded into this object or in its component object classes. A

human object initiates 16 segment objects and 15 joint objects. The *Joint* objects manage joint vertices trivially and may be removed for computational speed. *Segment* objects have a *TriangleFaceSet* object, which reads polygon vertices from a file for that segment shape. Segment shapes are originally parts of a VRML object that is downloaded from the Internet. These shapes may be drawn in wire-frame or smooth shaded modes. A *Box* object is defined for the outer bound of each shape. Box objects are used for two purposes: drawing stick figures and detecting mouse clicks.

The hierarchy between segment objects is hard coded in the Human class (Figure 21). Notice that this structure is static. If dynamic hierarchy management is required, a new class should handle Figure 21 as a tree structure.

```

void Human::draw()
{
    // Root
    drawRoot();
    glPushMatrix();
    // Body
    drawBody();
    glPushMatrix();
        // Neck and Head
        drawHead();
        glPopMatrix();
        glPushMatrix();
        // LeftArm
        drawLeftArm();
        glPopMatrix();
        glPushMatrix();
        // RightArm
        drawRightArm();
        glPopMatrix();
        glPopMatrix();
        glPushMatrix();
        // LeftLeg
        drawLeftLeg();
        glPopMatrix();
        glPushMatrix();
        // RightArm
        drawRightLeg();
        glPopMatrix();

    return;
}

```

Figure 21: Segment Hierarchy

Each sub *draw* function in Figure 21 calls *draw* functions of individual segments in sequential order as in Figure 22.

```
void Human::drawLeftArm(){
    // Right Upper Arm
    segment[L_UPPER_ARM]->draw(points, normals);

    // Right Fore Arm
    segment[L_FORE_ARM]->draw(points, normals);

    // Right Hand
    segment[L_HAND]->draw(points, normals);
    return;
}
```

Figure 22: Drawing Left Arm

Figure 23 shows how segment objects draw.

```
void Segment::draw( GLfloat ** points, GLfloat ** normals )
{
    // make translation and orientation to set posture
    localMatrix ->applyToCurrentMatrix();

    //draw model
    switch( modelType ){
        case STICK : ...
            stick ->draw();
            break;

        case SKIN : ...
            shape ->drawTriangles( points, normals );
            break;

        case WIRE_FRAME : ...
            shape ->drawTriangles( points, normals );
            break;
        default : ...
            break;
    }
    return;
}
```

Figure 23: Draw Function of Segment Class

The main focus of this research is the *localMatrix* object, which is an *H_Matrix* object. This is a component object of the Segment class. Each segment object's orientation and position are defined in *localMatrix* objects. The *H_Matrix* class is actually an abstract class

and can't be instantiated. Joint transformation matrices for 1 and 3 DOF (eq.4.3-6) are hard coded in different classes that are inherited from *H_Matrix* class. The reasons for this implementation are introduced in Chapters 3 and Chapter 4. The hand, upperArm, foot, upperLeg, head, neck and upperBody objects have *H_Matrix3DOF* objects, while the lowerArm and lowerLeg objects have *H_Matrix1DOF* objects. In general, these objects supply conversion functions for three types of rotation methods: Euler angles, quaternions, and vector-angle pair. Conversions are supplied from these representations to homogeneous matrices and vice versa. Direct conversion between quaternion and vector-angle pair representations is also supported. For quaternion algebra, the *H_Matrix* class uses a Quaternion object interface.

B. USER INPUTS

Double clicking on a segment switches to Euler angle inputs and draws a gimbal mechanism. Current Euler angles are read from an *H_Matrix* object for initialization of the corresponding gimbal mechanism. This mechanism has 1 or 3 Circle class objects. Dragging these circles changes Euler angles. These circles demonstrate gimbal lock and show joint angle limits that are important when quaternion inputs are converted.

The second input type, quaternion inputs, are entered by a mouse, which is read as vector-angle pairs. The vector is always perpendicular to the screen. Rotating a scene actually rotates this vector. Thus this vector is calculated by inverting the scene orientation matrix (Figure 24).

```

GLdouble

    cx =cos( -viewRotX * DEG_TO_RAD ),
    sx =sin( -viewRotX * DEG_TO_RAD ),
    cy =cos( -viewRotY * DEG_TO_RAD ),
    sy =sin( -viewRotY * DEG_TO_RAD );

// construct rotation vector from study angle
orientation[X] =(GLfloat) ( cx*sy );
orientation[Y] =(GLfloat) ( -sx );
orientation[Z] =(GLfloat) ( cx*cy );

```

Figure 24: Calculating the Vector that is Perpendicular to the Screen

The Z-axis is directed to out of screen in OpenGL coordinates. Thus, the last column of the inverse matrix gives the current Z-axis relative to the initial coordinate frame. Segment angles in 2D can be tracked in Window coordinates. A segment's joint center is the peak vertex of the angle, which can be easily found in the transformation matrix of that segment. A segment angle is measured between a mouse clicked point to a mouse released point. By using these three points, the cosine theorem is applied to calculate angles. Actually, joint centers are in world coordinates, while the other two points are in window coordinates. 3DCursor objects are used to project and unproject these points to draw 3D cursors and to calculate angles. The code of Figure 25 is run when the mouse button is released.

```

void UserControl::quaternionMotion(const GLint WIN_X, const GLint WIN_Y)
{
    GLfloat
    oldX =(GLfloat)selectionMark ->getWorldX(),
    oldY =(GLfloat)selectionMark ->getWorldY(),
    newX =(GLfloat)cursor3D ->getWorldX(),
    newY =(GLfloat)cursor3D ->getWorldY();

// gets angle between selected mouse coord. & joint & current mouse coord
orientation[3] =getAngleFm3Points(selectedSegment_jx,selectedSegment_jy,
                                oldX, oldY,
                                newX, newY);

// rotate segment with this rot. ang. & vec.
human ->modifyPosture( VECTOR_ANGLE, selectedSegment, orientation );
}

```

Figure 25: Response to Quaternion Input

The *modifyPosture* method of a human object calls other methods which ends with calling the *rotate* method of *H_Matrix* (Figure 26). This method is called by a *VECTOR_ANGLE* parameter, because the actual input is a vector-angle pair. But the next method, *rotateByVecAng*, converts this input to a quaternion to apply rotation. As Figure 13 demonstrates, applying vector-angle pair rotations to existing orientations is accomplished more efficiently by using quaternion algebra than by calling the *glRotate3f* method that leads to matrix construction and matrix multiplication.

```

boolean H_Matrix::rotate( const ROTATION_METHODS method,
                           const GLfloat* orientation)
{
    boolean rotationAccepted =FALSE;

    switch( method ){
        case VECTOR_ANGLE:
            rotationAccepted =rotateByVecAng( orientation );
            ...
    }
    return rotationAccepted;
}

boolean H_Matrix::rotateByVecAng( const GLfloat * orientation )
{
    boolean rotationAccepted =FALSE;

    Quaternion rotation( orientation );

    // apply rotation on existing quaternion orientation
    Quaternion newOrientation =rotation * quaternion;

    //checks if rotation in boundaries
    if( isRotationAcceptable( newOrientation ) ){

        quaternion =newOrientation;

        quatToMatrix();

        rotationAccepted =TRUE;// if it is, return accepted
    }

    return rotationAccepted;
}

```

Figure 26: Rotation of an Existing Orientation with a Vector-Angle Pair

C. CONSTRAINTS

As seen in Figure 26, the `rotateByVecAng` function calls the `isRotationAccepted` function by the new quaternion object. There is no way to constraint a quaternion, so the new orientation is converted to Euler angles by using homogeneous matrix conversion and vice versa (Figure 27). This causes additional computation at run-time.

```
boolean H_Matrix3DOF::isRotationAcceptable( QuaternionR & newOrientation )
{
    boolean accepted =FALSE; //default, don't accept rotation

    H_Matrix3DOF tmp;
    ...
    tmp.quaternion = newOrientation;
    tmp.quatToMatrix();
    tmp.matrixToEuler();

    if( tmp.isRotationAcceptable( tmp.angle ) ){ accepted =TRUE; }

    return accepted;
}
```

Figure 27: Constraints for Quaternion Representation

D. MOTION TRACKING

This program supports forward kinematics for 16 body segments to draw a human figure. Each segment multiply its local homogeneous matrix with the current model view matrix in the graphics pipeline to obtain its posture in the world coordinates. There is no need for inverse kinematics to apply the inputs of an inertial sensor tracking system. Instead, quaternion filter outputs are directly applied to segments. The code of Figure 28 is to import the `Qaef` object that is described and implemented in [DUMA99].

But the `applyToCurrentMatrix` function of `H_Matrix` class should be changed as in Figure 29 for the quaternion filter outputs. Because the quaternion filter produces unit quaternions in earth coordinates that are independent from previous joint transformations. But the joint position is still effected by previous joint transformations.

```

void SensorSystem::trackSegment()
{
    Quaternion vecAng = (q1 ->getResult()).toAxisAngles();
    orientation[X] =vecAng.getX();
    orientation[Y] =vecAng.getY();
    orientation[Z] =vecAng.getZ();
    orientation[3] =vecAng.getW();

    human -> setPosture( VECTOR_ANGLE, R_UPPER_ARM, orientation );

    vecAng = (q2 ->getResult()).toAxisAngles();
    orientation[X] =-vecAng.getX();
    orientation[Y] =vecAng.getY();
    orientation[Z] =-vecAng.getZ();
    orientation[3] =vecAng.getW();

    human -> setPosture( VECTOR_ANGLE, R_FORE_ARM, orientation );
}

```

Figure 28: Sensor Tracking Method

```

void H_Matrix::applyToCurrentMatrix (GLfloat view[16])
{
    GLfloat tempMatrix[16];
    //position is effected by parent joints transformation and camera motions
    // At this point, all parent joints transformations & camera motions is already
    // applied to graphic pipeline
    glTranslatef( joint_x, joint_y, joint_z );

    //hold transformed joint position
    glGetFloatv( GL_MODELVIEW_MATRIX, tempM );
    GLfloat
        transformed_x = tempMatrix [12],
        transformed_y = tempMatrix [13],
        transformed_z = tempMatrix [14];

    //orientation is not effected by parent joints transformations
    // (defined in earth coord., stored in joint_matrix[] )
    //but, orientation is also effected by camera motions
    tempMatrix [12] = tempMatrix [13] = tempMatrix [14] =0;
    for( GLint i=0; i<12; i++ )
        tempMatrix[i] =joint_matrix[i];
    glLoadMatrixf( view );
    glMultMatrixf( tempMatrix );

    //construct result transformation matrix
    glGetFloatv( GL_MODELVIEW_MATRIX, tempMatrix );
    tempMatrix [12] =transformed_x;
    tempMatrix [13] =transformed_y;
    tempMatrix [14] =transformed_z;

    //apply result transformation matrix
    glLoadMatrixf( tempMatrix );
}

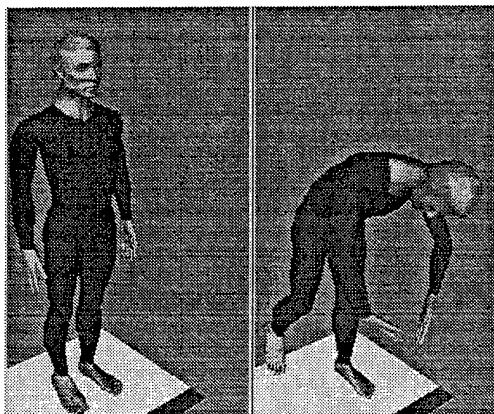
```

Figure 29: Construction of the Segment Transformation Matrix

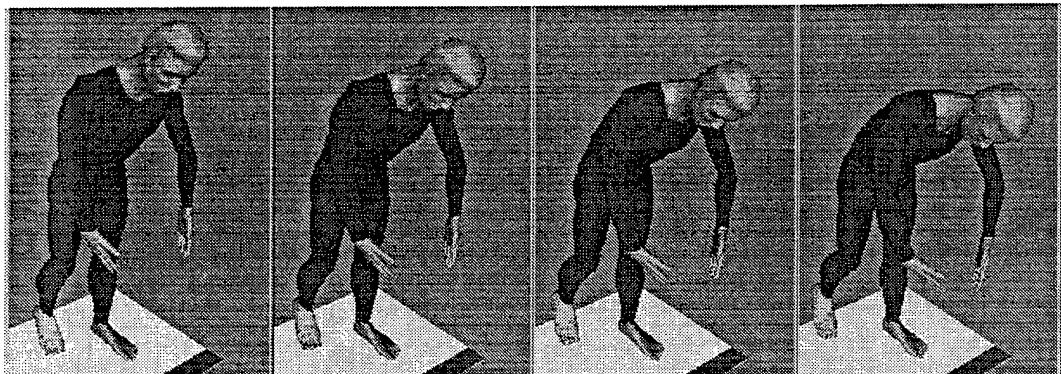
The code given in Figure 29 is different from the code that is given in Appendix B. Inputs may be checked by angle constraints, which causes extra conversion computations. Quaternions are applied to the graphics pipeline directly by using Eq.(4.42), which is efficient. The quaternion system doesn't use any trigonometric function and requires just 16 matrix multiplications.

E. RESULTS

For interpolating quaternions, Eq.(4.45) is also implemented. As stated in Chapter IV, there is no singularity with this interpolation. This algorithm can be used for dead reckoning of human segments in large scale networked virtual environments. Figure 30 shows two key frames and the in-between frames of interpolation.



(a) User Defined Key Frames



(b) Computed In-between Frames

Figure 30: Quaternion Interpolation

Algebraic inverse kinematics solutions that are given in Chapter V are also coded in the ***InverseKinematics*** class, which is a component object of the **UserControl** class. But details and accuracy are not the focus of this research. Figure 31 is an example for hand and foot motions.

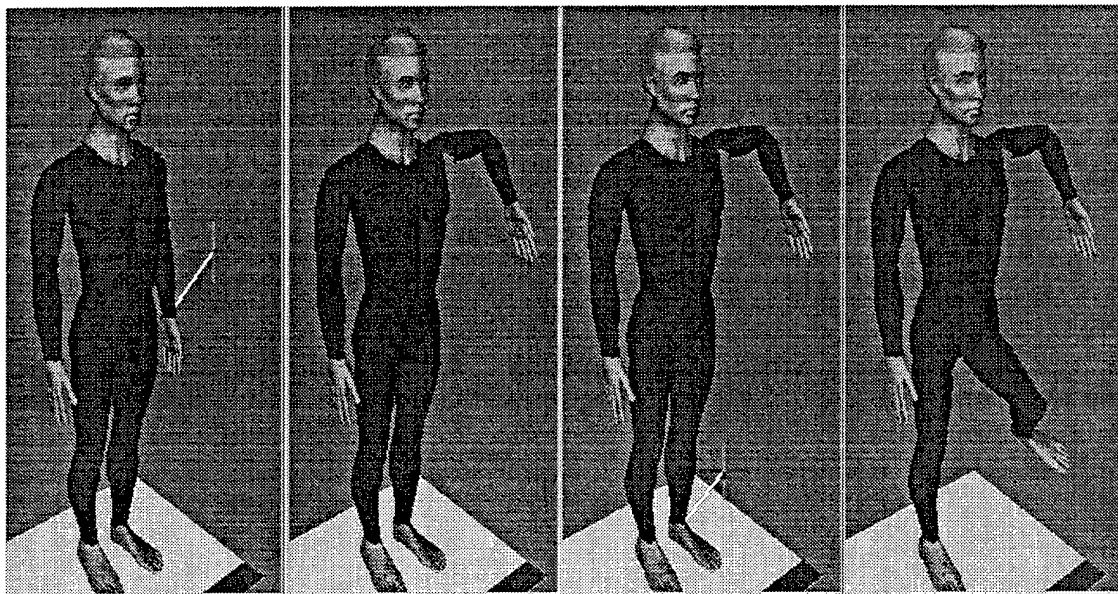
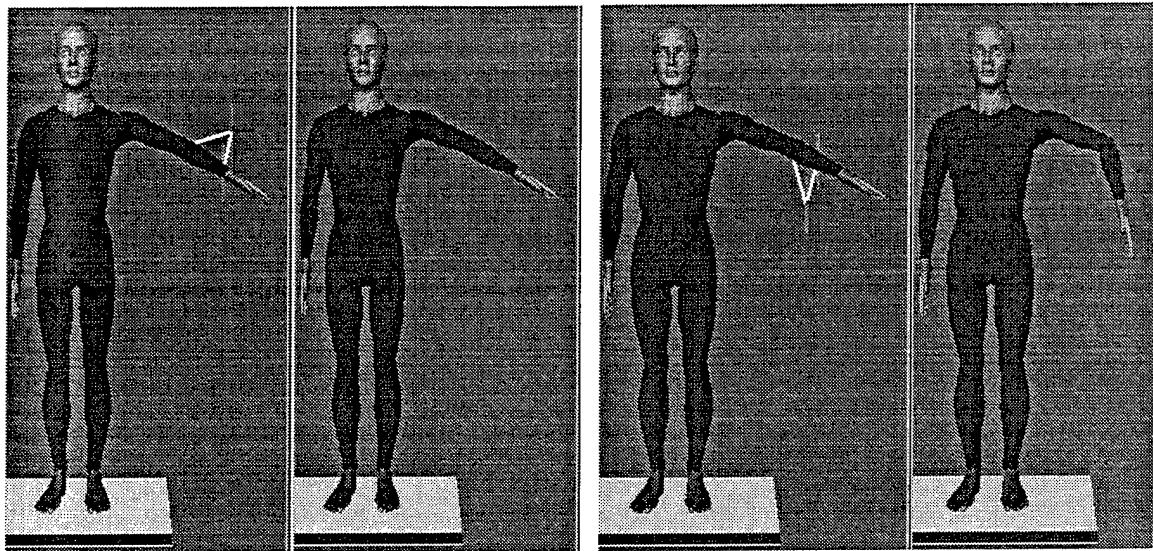


Figure 31: The Left Hand and the Left Foot Motions by Inverse Kinematics

The main result of this study is the demonstration of a human articulated body model with quaternions, which allows efficient computation and eliminates the singularity problems of Euler angles. The joint constraints are also applied to the model when quaternion and Euler angle inputs are entered by a mouse. Figure 32 shows two mouse inputs for elbow. First one force the elbow to make an impossible motion. The second is a normal elbow rotation. The program rejects the first motion.



(a) Impossible Motion for Elbow

(b) Accepted Motion for Elbow

Figure 32: Demonstration of Joint Constraints

Figure 33 shows six frames of walking as a procedural animation.

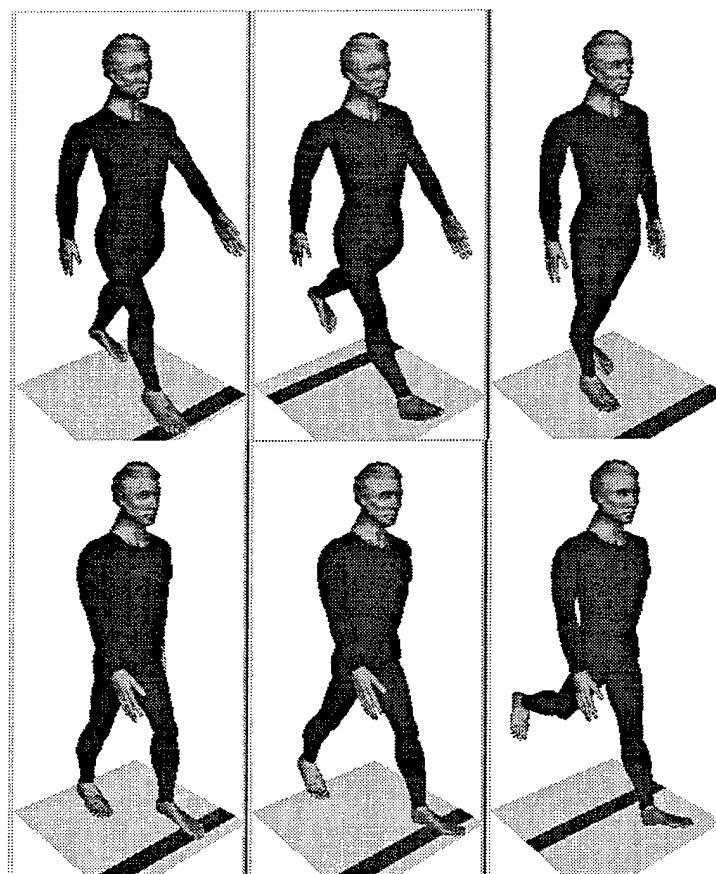
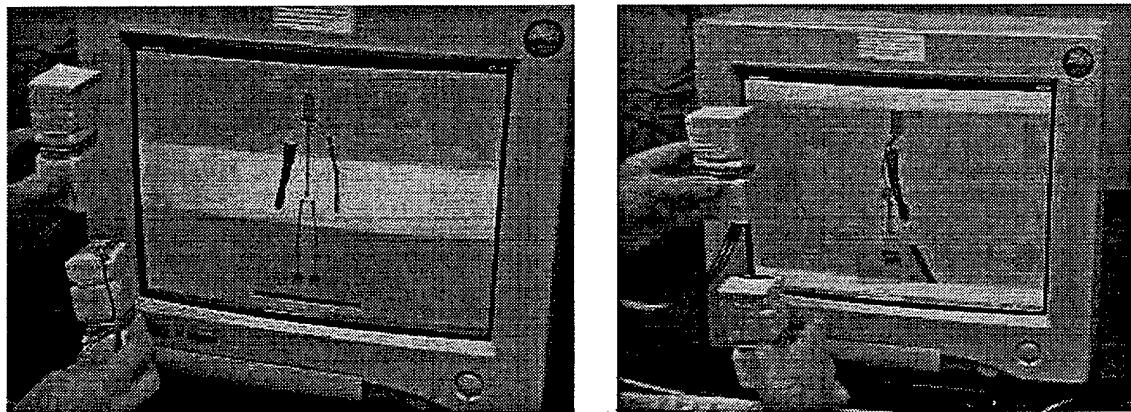
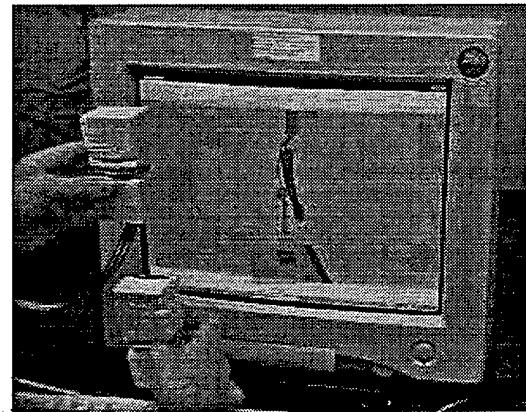


Figure 33: Walking as a Procedural Animation

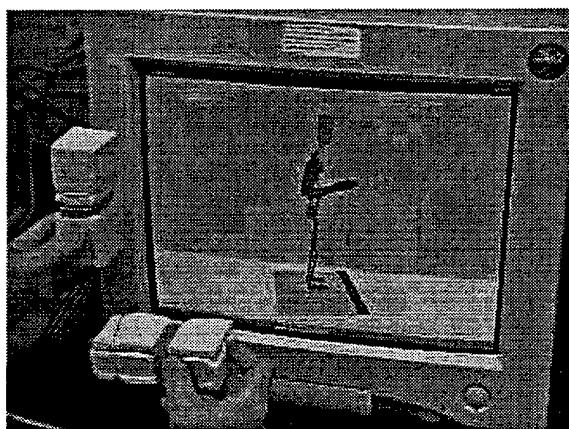
Figure 34 shows the right arm motion tracking with two inertial sensor inputs for the right shoulder and the right elbow. The quaternion attitude filter filters the inputs.



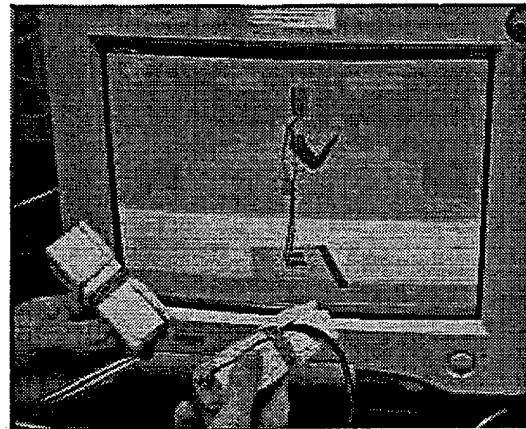
(a) Initial Posture



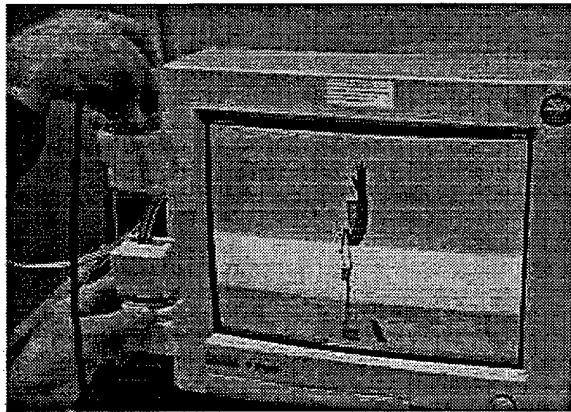
(b) Initial Posture from another Point of View



(c) Elbow Motion



(d) Elbow and Shoulder motion



(e) Shoulder has 90 degrees elevation (No singularity)

Figure 34: The Motion Tracking of Right Shoulder and Right Elbow with two Inertial Sensors and the Quaternion Attitude Filter

The TriangleFaceSet class reads data from the given file. Thus, replacing human segment shapes are easy, when new vertices are calculated. The object-oriented design of the model also facilitates new implementations for different purposes.

VII. SUMMARY AND CONCLUSION

A. SUMMARY

This study presents a human articulated rigid body with quaternions that can be used for real-time computer graphics applications in Virtual Environments (VE). There are many issues for modeling human figures. New methods are developing for more realism and automatic control of the figures. But, the real-time requirements usually constrain the animation to consist of flat or Gouraud-shaded polygons with texture mapping [WATT92]. For real-time systems, the joint motions still must be realistic after making a high level of abstraction for the appearance. The articulated structure is the most important part of the human figure to define the realistic postures. Computation and the behavior of the articulated structure and the input devices are the case for real-time human figure simulation systems.

There are two different approaches to construct the human articulated structure. In robotics and computer graphics, many applications use MDH notation to build articulated structures [SKOP97]. A different method uses only segment orientations represented by Euler angles to make the postures of the human figure [FREY96]. Chapter 3 of this thesis discusses the efficiency issues of these two methods. With the second method, each segment joint of the human figure has a transformation matrix. This study brings a new approach to represent the transformations of joints by using the quaternions as the state vectors of the joints.

When quaternions are used, it is possible to rotate vertices of segment shapes and the positions of child joints without using homogeneous matrices. But, today's most common graphic engines use homogeneous matrices, and the author chose to construct joint

transformation matrices for rendering purposes. Nevertheless, using quaternions is still powerful for many computations. The most important superiority of the quaternion representation is that it is independent of coordinate axes and shows no singularity at any value. Besides that, making consecutive rotations on existing orientation necessitates less computation than making matrix multiplications. The other method, Euler angles, is inadequate for interpolation and should be converted other forms for interpolation purposes. This adds extra computation at run time.

The program that is developed in this study can be used for the experimental purposes, which demonstrates Euler angle and quaternion methods and lets users define posture with forward and inverse kinematics. Joint constraints are applied to the mouse inputs. The real-time display of human arm tracking with two inertial sensors, key-frame animation, and walking are the other features of the program. Other human figures can also be used by changing joint positions and segment vertex data in the vertex files. The user manual of the program is given in Appendix A.

B. CONCLUSIONS AND FUTURE RESEARCH

Chapter IV of this thesis compares quaternion and Euler angle methods for many considerations. The quaternion method solves singularity problems of the Euler angle methods. It is also more powerful than the Euler angle method when frequent renormalizations of rotational operators are needed. On the other hand, Euler angle method is the only solution to add constraint to joint angles. When quaternions are used as the state vector of the articulated structure, PDU packets are longer than those constructed by Euler angles. Today, there are many graphic hardware implementations that support matrix algebra, which reduces the computational expense of the Euler angle

method. Using one or the other method depends on the application purposes. Both methods may be used within the same software for the best performance.

An important future research is to combine this study with [DUMA99] that introduces quaternion attitude filter. This filter is designed to compensate many problems that are encountered with Euler angle methods and produces unit quaternions as orientation data. These outputs can be used to track human segments. The human model that is introduced in this thesis can define human postures with quaternion data. A body suit that has 15 inertial motion tracker sensor that each uses a quaternion attitude filter can track a whole human body.

APPENDIX A: USER MANUAL

The program developed for this thesis runs only on Windows95/98 and WindowsNT platforms. Glut library is used for window specific tasks and OpenGL is used for 3D modeling and rendering. The main interaction device of the program is a mouse. Keystrokes are also processed as hot keys for key frame and procedural animations. Pop-up menus are the only user interface to switch between different modes of the program. The main pop-up menu is reached when the right mouse button is pressed. The other pop-up menus are the sub menus of the main menu.

A. NAVIGATION MODE

When the left mouse button is pressed in the screen space and the mouse is dragged, the mouse motions cause the camera motions. There are 3 different navigation modes of the program. The *Walk* mode translates the camera position on 3 axes. The *Pan* mode translates the camera position only on OpenGL X and OpenGL Y axes (Horizontal and vertical axes of the screen). The *Study* mode rotates the camera on 3 axes. These three navigation type can be chosen from the first sub menu of the main menu.

B. MODEL TYPE

The second sub menu lets user to chose the model type of the human. It can be a *Stick* model or a *Wire Frame* model or a *Smooth Shaded* model.

C. POSTURE CONTROL MODE

There are 3 types of posture control mode: *inertial sensor tracking*, *mouse control*, and *procedural animation*. *Key frames* can be set and can be interpolated when the posture control mode is the mouse control. These modes are selected from the third sub menu.

1. Inertial Sensor Tracking

There must be two inertial sensors that are connected to the system to switch to this option. The program code that is given in Appendix B should also be modified as in Figure 29. For real-time concerns, the model type should be stick model or the code may be modified to draw only the arm as smooth shaded polygons and the rest of the body as stick figure.

2. Mouse Control

A selection occurs when the left mouse button is pressed while the cursor is on a segment or on a circle of the gimbal mechanism.

a. *Forward Kinematics*

There are two options to change the orientation of the selected segment. If the left mouse button is released after a segment selection without changing the mouse cursor position, a gimbal system appears on the top left corner of the screen. Each circle on the **gimbal system** represents a DOF of the selected segment (Knees and elbows have only 1 DOF, and the all other segments have 3 DOF). At this point, mouse drags do not cause the camera motions. If any circle is selected, the mouse drags cause the segment and the circle rotations. Rotations are restricted by joint constraints. The left mouse button should be pressed and released on the same segment to quit from this option.

As a second option after a segment selection, the mouse is dragged in any direction to make a 2D angle and then released to apply a **quaternion rotation**. If this rotation is accepted by joint constraints, the segment change its posture as defined. The detailed explanation of this option is given in Chapter IV.

b. Inverse Kinematics

In this mode, user can select only feet and hands. User can define a goal position for the selected end-effector by dragging mouse. If the motion is acceptable, the program calculates in-between joint orientations and applies to reach to the goal position.

3. Procedural Animation

User can apply walking procedures to the human figure. There are 2 types of walking procedures, which are defined by inverse kinematics and forward kinematics. The up and down arrows are used to speed up and slow down the walking procedures.

4. Key Frame Animation

Key frames are defined by mouse control. User can add, insert, and remove key frames from a linked list either by keystrokes or by sub menu item selections. It is possible to switch between key frames. When all keys are set, run option makes the interpolation.

APPENDIX B: 3D HUMAN FIGURE SIMULATION SOFTWARE

| | |
|----------------------------|-----|
| Box.h..... | 86 |
| Box.cpp..... | 87 |
| Circle.h..... | 91 |
| Circle.cpp..... | 92 |
| Cursor3D.h..... | 94 |
| Cursor3D.cpp..... | 95 |
| Floor.h..... | 98 |
| Floor.cpp..... | 99 |
| GimbalSystem.h..... | 101 |
| GimbalSystem.cpp..... | 102 |
| GlutBaseClass.h..... | 104 |
| GlutBaseClass.cpp..... | 106 |
| H_Matrix.h..... | 116 |
| H_Matrix.cpp..... | 117 |
| H_Matrix1DOF.h..... | 123 |
| H_Matrix.1DOF.cpp..... | 124 |
| H_Matrix.3DOF.h..... | 128 |
| H_Matrix.3DOF.cpp..... | 129 |
| Human.h..... | 132 |
| Human.cpp..... | 134 |
| InverseKinematics.h..... | 142 |
| InverseKinematics.cpp..... | 143 |
| Joint.h..... | 147 |
| Joint.cpp..... | 148 |
| KeyFrameAnim.h..... | 152 |
| KeyFrameAnim.cpp..... | 153 |
| Posture.h..... | 158 |
| Posture.cpp..... | 159 |
| ProceduralAnim.h..... | 160 |
| ProceduralAnim.cpp..... | 161 |
| QuaternionR.h..... | 167 |
| QuaternionR.cpp..... | 168 |
| Segment.h..... | 172 |
| Segment.cpp..... | 174 |
| SensorSystem.h..... | 179 |
| SensorSystem.cpp..... | 180 |
| TriangleFaceSet.h..... | 181 |
| TriangleFaceSet.cpp..... | 182 |
| UserControl.h..... | 185 |
| UserControl.cpp..... | 187 |
| utility.h..... | 194 |
| utility.cpp..... | 196 |

```

***** : Box.h
// FILE : Box.h
// DESCRIPTION: Used for bounding boxes and stick models
*****  

#ifndef __BOX_H_
#define __BOX_H_  

#include <GL/glut.h>
#include "utility.h"  

enum BOX_VOLUME_TERMS {
    BOTTOM_PLATE, TOP_PLATE, TWO_PLATES, THREE_POINTS, FOUR_POINTS
};  

class Box{
public :-----PUBLIC-----  

    //CONSTRUCTORS
    Box( const GLfloat END_X,
        const GLfloat END_Y,
        const GLfloat END_Z );
    Box( const GLfloat END_Y,
        const GLfloat POS_X,
        const GLfloat NEG_X,
        const GLfloat POS_Z,
        const GLfloat NEG_Z );
    Box( GLfloat vol[TWO_PLATES][FOUR_POINTS][THREE_D] );
    Box( Box & );  

    //DESTRUCTOR
    ~Box();  

    //OPERATORS
    Box& operator=( const Box & );  

    //FUNCTIONS
    void draw();
    void show();
    GLfloat getHeight();  

private :-----PRIVATE-----  

    //VARIABLEs
    GLfloat
        volume[TWO_PLATES][FOUR_POINTS][THREE_D],
        height;  

};  

#endif

```

```

//*****
// FILE    : Box.cpp
// DESCRIPTION: implementation of bounding boxes and stick figures
//*****


#include "Box.h"

//-----
Box::Box( GLfloat vol[TWO_PLATES][FOUR_POINTS][THREE_D] )
{
    for( GLint plate=0; plate<TWO_PLATES; plate++ ){

        for( GLint point=0; point<FOUR_POINTS; point++ ){

            for( GLint axis=0; axis<THREE_D; axis++ ){

                volume[plate][point][axis] =vol[plate][point][axis];
            }
        }
    }

//-----
Box::Box( const GLfloat END_X,
          const GLfloat END_Y,
          const GLfloat END_Z )
{
    static const GLfloat  DELTA_X =0.08f,
                           DELTA_Z =0.05f;
    #if 0
    GLint    plate1 =BOTTOM_PLATE,
            plate2 =TOP_PLATE;

    if( END_Y < 0 ) {

        plate1 =TOP_PLATE;
        plate2 =BOTTOM_PLATE;
    }

    for( GLint point=0; point<FOUR_POINTS; point++ ){

        volume[plate1][point][Y] =0;
        volume[plate2][point][Y] =END_Y;
    }

    volume[plate1][0][X] =-DELTA_X;
    volume[plate1][0][Z] =-DELTA_Z;
    volume[plate1][1][X] =-DELTA_X;
    volume[plate1][1][Z] =DELTA_Z;
    volume[plate1][2][X] =DELTA_X;
    volume[plate1][2][Z] =DELTA_Z;
    volume[plate1][3][X] =DELTA_X;
    volume[plate1][3][Z] =-DELTA_Z;

    volume[plate2][0][X] =END_X-DELTA_X;
    volume[plate2][0][Z] =END_Z-DELTA_Z;

```

```

volume[plate2][1][X] =END_X-DELTA_X;
volume[plate2][1][Z] =END_Z+DELTA_Z;
volume[plate2][2][X] =END_X+DELTA_X;
volume[plate2][2][Z] =END_Z+DELTA_Z;
volume[plate2][3][X] =END_X+DELTA_X;
volume[plate2][3][Z] =END_Z-DELTA_Z;

height =( END_Y > 0 ) ? END_Y : -END_Y;
}

//-----
Box::Box( const GLfloat END_Y,
          const GLfloat POS_X,
          const GLfloat NEG_X,
          const GLfloat POS_Z,
          const GLfloat NEG_Z )
{
    GLint plate1 =BOTTOM_PLATE,
          plate2 =TOP_PLATE;

    if( END_Y < 0 ) {

        plate1 =TOP_PLATE;
        plate2 =BOTTOM_PLATE;
    }

    for( GLint point=0; point<FOUR_POINTS; point++ ){

        volume[plate1][point][Y] =0;
        volume[plate2][point][Y] =END_Y;
    }

    volume[plate1][0][X] =NEG_X;
    volume[plate1][0][Z] =NEG_Z;
    volume[plate1][1][X] =NEG_X;
    volume[plate1][1][Z] =POS_Z;
    volume[plate1][2][X] =POS_X;
    volume[plate1][2][Z] =POS_Z;
    volume[plate1][3][X] =POS_X;
    volume[plate1][3][Z] =NEG_Z;

    volume[plate2][0][X] =NEG_X;
    volume[plate2][0][Z] =NEG_Z;
    volume[plate2][1][X] =NEG_X;
    volume[plate2][1][Z] =POS_Z;
    volume[plate2][2][X] =POS_X;
    volume[plate2][2][Z] =POS_Z;
    volume[plate2][3][X] =POS_X;
    volume[plate2][3][Z] =NEG_Z;

    height =( END_Y > 0 ) ? END_Y : -END_Y;
}

```

```

//-----
Box::Box( Box & box )
{
    // INITIALIZE
    for( GLint plate=0; plate<TWO_PLATES; plate++ ){

        for( GLint point=0; point<FOUR_POINTS; point++ ){

            for( GLint axis=0; axis<THREE_D; axis++ ){

                volume[plate][point][axis] =box.volume[plate][point][axis];
            }
        }
    }

    height =box.height;
}

//-----
Box::~Box()
{
}

//-----
Box& Box::operator=( const Box& box )
{
    for( GLint plate=0; plate<TWO_PLATES; plate++ ){

        for( GLint point=0; point<FOUR_POINTS; point++ ){

            for( GLint axis=0; axis<THREE_D; axis++ ){

                volume[plate][point][axis] =box.volume[plate][point][axis];
            }
        }
    }

    return (*this);
}

//-----
void Box::draw()
{
    static const GLint
        FACES =6,
        INDICES[FACES][FOUR_POINTS][2] ={



{{ BOTTOM_PLATE, 0 },{ BOTTOM_PLATE, 3 },{ BOTTOM_PLATE, 2 },{ BOTTOM_PLATE, 1 }}, //0
{{ TOP_PLATE, 0 },{ TOP_PLATE, 1 },{ TOP_PLATE, 2 },{ TOP_PLATE, 3 } }, //1
{{ BOTTOM_PLATE, 0 },{ BOTTOM_PLATE, 1 },{ TOP_PLATE, 1 },{ TOP_PLATE, 0 } }, //2
{{ BOTTOM_PLATE, 1 },{ BOTTOM_PLATE, 2 },{ TOP_PLATE, 2 },{ TOP_PLATE, 1 } }, //3
{{ BOTTOM_PLATE, 2 },{ BOTTOM_PLATE, 3 },{ TOP_PLATE, 3 },{ TOP_PLATE, 2 } }, //4
{{ BOTTOM_PLATE, 3 },{ BOTTOM_PLATE, 0 },{ TOP_PLATE, 0 },{ TOP_PLATE, 3 } } //5
};
}

```

```

GLfloat
normals[FACES][3] ={
    { 0,-1,0 }, { 0,1,0 }, { -1,0,0 }, { 0,0,1 }, { 1,0,0 }, { 0,0,-1 }
};

for( GLint face =0; face<FACES; face++ ){

    glBegin(GL_POLYGON);
        glNormal3fv( normals[face] );
        for( GLint point=0; point<FOUR_POINTS; point++ ){

            glVertex3fv( volume[INDICES[face][point][0]][INDICES[face][point][1]] );
        }
    glEnd();
}

//-----
void Box::show()
{
    glBegin(GL_QUAD_STRIP);
    for(GLint i=0; i<FOUR_POINTS; i++){

        glVertex3fv(volume[TOP_PLATE][i]);
        glVertex3fv(volume[BOTTOM_PLATE][i]);
    }
    glVertex3fv(volume[TOP_PLATE][0]);
    glVertex3fv(volume[BOTTOM_PLATE][0]);
    glEnd();

    return;
}

//-----
GLfloat Box::getHeight()
{
    return height;
}

```

```

//*****: Circle.h
// FILE : Circle.h
// DESCRIPTION: Used for circles of Gimbal Mech.
//*****: Circle.h

#ifndef __CIRCLE_H_
#define __CIRCLE_H_

#include <GL/glut.h>
#include "utility.h"

class Circle{

public :-----P U B L I C-----


    //CONSTRUCTORS
    Circle( GLint );    //default
    Circle( Circle & ); //copy

    //DESTRUCTOR
    ~Circle();

    //FUNCTIONS
    void draw();
    void increment();
    void decrement();

    GLfloat getAngle();
    void setAngle( GLfloat );

private :-----P R I V A T E-----


    //CONSTs
    static const GLfloat DELTA;

    //OPERATORS
    Circle& operator=( const Circle & );

    //VARIABLES
    GLfloat radius,
            angle;

};

#endif

```

```

//*****
// FILE    : Circle.cpp
// DESCRIPTION:
//*****

#include "Circle.h"

//*****INITIALIZE STATIC DATA MEMBERS *****
const GLfloat Circle::DELTA =5;

//*****END STATIC DATA MEMBER INITIALIZATION *****

//-----
Circle::Circle( GLint axis ){

    // INITIALIZE
    angle =0;

    switch( axis ){

        case 0:radius =6;
                  break;
        case 1:radius =8;
                  break;
        case 2:radius =10;
                  break;
        default:break;
    }
}

//-----
Circle::~Circle(){

}

//-----
void Circle::increment(){

    angle +=DELTA;
    if(angle > 360) angle -=360;
}

//-----
void Circle::decrement(){

    angle -=DELTA;
    if(angle < 0) angle +=360;
}

//-----
GLfloat Circle::getAngle(){

    return angle;
}

```

```
//-----  
void Circle::setAngle( GLfloat a ){  
  
    angle =a;  
}  
  
//-----  
void Circle::draw( )  
{  
    if( angle != OUT_RANGE ){  
  
        glRotatef( angle, 0, 1, 0 );  
        glutSolidTorus( 0.5f, radius, 5 ,20 );  
        //cones  
        glPushMatrix();  
        //upper cone  
        glTranslatef( 0, radius, 0 );  
        glRotatef( -90, 1, 0, 0 );  
        glutSolidCone( 0.8, 2, 6, 1 );  
        //lower cone  
        glTranslatef( 0, 0, -2 * radius );  
        glRotatef( 180, 1, 0, 0 );  
        glutSolidCone( 0.8, 2, 6, 1 );  
        glPopMatrix();  
    }  
    return;  
}
```

```

//*****
// FILE    : Cursor3d.h
// DESCRIPTION: Used for creating 3D cursor effects for tracking
//*****


#ifndef __CURSOR3D_H__
#define __CURSOR3D_H__


#include <GL/glut.h>
#include "utility.h"

class Cursor3D{

public ://-----P U B L I C-----


    //CONSTRUCTORS
    Cursor3D();                                //default
    Cursor3D( Cursor3D & ); //copy

    //DESTRUCTOR
    ~Cursor3D();


    //FUNCTIONs
    void setWindowCoord( const GLint WIN_X,
                         const GLint WIN_Y,
                         const GLint WIN_Z );
    void setWindowCoord( const GLint WIN_X, const GLint WIN_Y );
    void setWorldCoord( const GLdouble WORLD_X,
                        const GLdouble WORLD_Y,
                        const GLdouble WORLD_Z );

    GLdouble getWorldX();
    GLdouble getWorldY();
    GLdouble getWorldZ();

    void draw();

private ://-----P R I V A T E-----


    //OPERATORS
    Cursor3D& operator=( const Cursor3D & );

    //FUNCTIONs
    void drawCoordSystem();

    //OBJECTs

    //VARIABLEs
    GLdouble
        windowCoord[THREE_D],
        worldCoord[THREE_D];
};

#endif

```

```

//*****
// FILE    : Cursor3D.cpp
// DESCRIPTION:
//*****

#include <math.h>
#include "Cursor3D.h"

//-----
Cursor3D::Cursor3D()
{
    // INITIALIZE
    for( GLint axis =0; axis <THREE_D; axis++ ){

        worldCoord [ axis ] =0;
        windowCoord[ axis ] =0;
    }
}

//-----
Cursor3D::~Cursor3D()
{
}

//-
void Cursor3D::draw(){

    drawCoordSystem();
}

//-
void Cursor3D::drawCoordSystem( )
{
    static const GLfloat LENGTH =0.5f;

//worldCoord[X]worldCoord[Y]worldCoord[Z] lines
    glBegin(GL_LINES);
    glColor3f(0, 0.5f, 1);
    glVertex3d(worldCoord[X]-LENGTH,worldCoord[Y],worldCoord[Z]);
    glVertex3d(worldCoord[X]+LENGTH,worldCoord[Y],worldCoord[Z]);
    glColor3f(0, 1, 0);
    glVertex3d(worldCoord[X],worldCoord[Y]-LENGTH,worldCoord[Z]);
    glVertex3d(worldCoord[X],worldCoord[Y]+LENGTH,worldCoord[Z]);
    glColor3f(1, 0, 0);
    glVertex3d(worldCoord[X],worldCoord[Y],worldCoord[Z]-LENGTH);
    glVertex3d(worldCoord[X],worldCoord[Y],worldCoord[Z]+LENGTH);
    glEnd();

    return;
}

```

```

//-----
void Cursor3D::setWindowCoord( const GLint WIN_X,
                               const GLint WIN_Y,
                               const GLint WIN_Z )
{
    windowCoord[Z] =WIN_Z;

    setWindowCoord( WIN_X, WIN_Y );

    return;
}

//-----
void Cursor3D::setWindowCoord( const GLint WIN_X, const GLint WIN_Y )
{
    GLint viewPort[4];
    GLdouble modelViewMatrix[16], projectionMatrix[16];

    // read current viewPort, model and project matrix values
    glGetIntegerv( GL_VIEWPORT, viewPort );
    glGetDoublev( GL_MODELVIEW_MATRIX, modelViewMatrix );
    glGetDoublev( GL_PROJECTION_MATRIX, projectionMatrix );

    windowCoord[X] =WIN_X;
    windowCoord[Y] =viewPort[3] -WIN_Y -1;

    // window to world func.
    gluUnProject( windowCoord[X], windowCoord[Y], windowCoord[Z],
                  modelViewMatrix, projectionMatrix, viewPort ,
                  & worldCoord[X], & worldCoord[Y], & worldCoord[Z] );
    return;
}

//-----
void Cursor3D::setWorldCoord( const GLdouble WORLD_X,
                               const GLdouble WORLD_Y,
                               const GLdouble WORLD_Z )
{
    GLint viewPort[4];
    GLdouble modelViewMatrix[16], projectionMatrix[16];

    // read current viewPort, model and project matrix values
    glGetIntegerv( GL_VIEWPORT, viewPort );
    glGetDoublev( GL_MODELVIEW_MATRIX, modelViewMatrix );
    glGetDoublev( GL_PROJECTION_MATRIX, projectionMatrix );

    worldCoord[X] =WORLD_X;
    worldCoord[Y] =WORLD_Y;
    worldCoord[Z] =WORLD_Z;

    // world to window func.
    gluProject( worldCoord[X], worldCoord[Y], worldCoord[Z],
                modelViewMatrix, projectionMatrix, viewPort ,
                & windowCoord[X], & windowCoord[Y], & windowCoord[Z] );
    return;
}

```

```
//-----
GLdouble Cursor3D::getWorldX()
{
    return worldCoord[X];
}

//-----
GLdouble Cursor3D::getWorldY()
{
    return worldCoord[Y];
}

//-----
GLdouble Cursor3D::getWorldZ()
{
    return worldCoord[Z];
}
```

```

***** :Floor.h
// DESCRIPTION: Simulates floor fow walking
*****


#ifndef __FLOOR_H__
#define __FLOOR_H__


#include <GL/glut.h>
#include <Math.h>
#include "utility.h"
#include "ProceduralAnim.h"

class Floor{

public :-----P U B L I C-----


//CONSTRUCTORS
Floor();           //default
Floor( Floor & ); //copy

//DESTRUCTOR
~Floor();

//FUNCTIONs
void draw();
void slide();
void increaseFrameRate();
void decreaseFrameRate();


private :-----P R I V A T E-----


//CONSTs
static const GLfloat
    LENGTH, HEIGHT, BAND_WIDTH;

//OPERATORS
Floor& operator=( const Floor & );


//VARIABLEs
GLfloat
    frameAcc,
    frameRate,
    maxFrameRate,
    minFrameRate,
    bandZ;
};

#endif

```

```

//*****
// FILE    : Floor.cpp
// DESCRIPTION:
//*****

#include "Floor.h"

//*****INITIALIZE STATIC DATA MEMBERS *****
const GLfloat
    Floor::LENGTH  =2,
    Floor::HEIGHT  =-4.275f,
    Floor::BAND_WIDTH=0.5f;

//*****END STATIC DATA MEMBER INITIALIZATION *****

//-----
Floor::Floor(){

    // INITIALIZE
    frameRate =0.2f;
    bandZ    =0;

    GLfloat
        syncRatio =( (2*LENGTH)-BAND_WIDTH ) / PI;

    frameRate =frameAcc =syncRatio * ProceduralAnim.FRAME_ACC;
    maxFrameRate  =syncRatio * ProceduralAnim.MAX_FRAME_RATE;
    minFrameRate  =syncRatio * ProceduralAnim.MIN_FRAME_RATE;
}

//-----
Floor::~Floor(){

}

//-----
void Floor::draw( )
{
    static const GLfloat BAND_HEIGHT =0.015f;

    glDisable(GL_LIGHTING);

    //draw floor
    glColor3f(0.8f, 0.8f, 0.8f);
    glBegin(GL_POLYGON);
    glNormal3f( 0, 1, 0 );
    glVertex3f( -LENGTH, HEIGHT, -LENGTH );
    glVertex3f( -LENGTH, HEIGHT, LENGTH);
    glVertex3f( LENGTH, HEIGHT, LENGTH);
    glVertex3f( LENGTH, HEIGHT, -LENGTH);
    glEnd();
}

```

```

//draw band on the floor
glColor3f(1,0,0);
glBegin(GL_POLYGON);
    glNormal3f( 0, 1, 0 );
    glVertex3f( -LENGTH, HEIGHT+BAND_HEIGHT, LENGTH-BAND_WIDTH-bandZ );
    glVertex3f( -LENGTH, HEIGHT+BAND_HEIGHT, LENGTH-bandZ );
    glVertex3f( LENGTH, HEIGHT+BAND_HEIGHT, LENGTH-bandZ );
    glVertex3f( LENGTH, HEIGHT+BAND_HEIGHT, LENGTH-BAND_WIDTH-bandZ );
glEnd();

glEnable(GL_LIGHTING);
return;
}

//-----
void Floor::slide()
{
    bandZ += frameRate; //update animation const
    if( bandZ > (2*LENGTH)-BAND_WIDTH ) bandZ =0;
}

//-----
void Floor::increaseFrameRate()
{
    frameRate +=frameAcc;

    if( frameRate >maxFrameRate ) frameRate =maxFrameRate;

    return;
}

//-----
void Floor::decreaseFrameRate()
{
    frameRate -=frameAcc;

    if( frameRate <minFrameRate ) frameRate =minFrameRate;

    return;
}

```

```

//*****GIMBALSYSTEM.H*****
//FILE : GimbalSystem.h
//DESCRIPTION: Interaction tool for Euler inputs
//*****GIMBALSYSTEM.H*****


#ifndef __GIMBALSYSTEM_H__
#define __GIMBALSYSTEM_H__


#include <GL/glut.h>
#include "Utility.h"
#include "Circle.h"


class GimbalSystem{

public :-----P U B L I C-----


//CONSTRUCTORS
GimbalSystem();           //default
GimbalSystem( GimbalSystem & ); //copy

//DESTRUCTOR
~GimbalSystem();


//FUNCTIONs
void draw();
void drawMouseDetectors();
void increment( AXIS );
void decrement( AXIS );

void setAngle( GLfloat * const );
void getAngle( GLfloat * const );


private :-----P R I V A T E-----


//OPERATORS
GimbalSystem& operator=( const GimbalSystem & );


//OBJECTs
Circle *x_axis,
        *y_axis,
        *z_axis;
};

#endif

```

```

//*****
// FILE    : GimbalSystem.cpp
// DESCRIPTION:
//*****

#include "GimbalSystem.h"

//-----
GimbalSystem::GimbalSystem(){
    // INITIALIZE
    x_axis =new Circle(X);
    y_axis =new Circle(Y);
    z_axis =new Circle(Z);
}

//-----
GimbalSystem::~GimbalSystem(){
    delete x_axis;
    delete y_axis;
    delete z_axis;
}

//-----
void GimbalSystem::increment( AXIS axis ){

    switch(axis){
        case X :x_axis->increment();
                   break;
        case Y :y_axis->increment();
                   break;
        case Z :z_axis->increment();
                   break;
    }
}

//-----
void GimbalSystem::decrement( AXIS axis ){

    switch(axis){
        case X :x_axis->decrement();
                   break;
        case Y :y_axis->decrement();
                   break;
        case Z :z_axis->decrement();
                   break;
    }
}

//-----
void GimbalSystem::getAngle( GLfloat * const eulerAngle)
{
    eulerAngle[X] =x_axis ->getAngle();
    eulerAngle[Y] =y_axis ->getAngle();
    eulerAngle[Z] =z_axis ->getAngle();
    return;
}

```

```

//-----
void GimbalSystem::setAngle( GLfloat * const eulerAngle)
{
    x_axis ->setAngle( eulerAngle[X] );
    y_axis ->setAngle( eulerAngle[Y] );
    z_axis ->setAngle( eulerAngle[Z] );
}

//-----
void GimbalSystem::draw( )
{
    //NOSE (openGL....z-axis)
    glColor3f(0, 0, 1);
    glRotatef(90, 1, 0, 0);
    glRotatef(90, 0, 1, 0);
    z_axis->draw();

    //AZIMUTH (openGL....y-axis)
    glColor3f(0, 1, 0);
    glRotatef(-90, 0, 1, 0);
    glRotatef(-90, 1, 0, 0);
    y_axis->draw();

    //ELEVATION (openGL....x-axis)
    glColor3f(1, 0, 0);
    glRotatef(90, 1, 0, 0);
    glRotatef(-90, 0, 0, 1);
    x_axis->draw();
}

//-----
void GimbalSystem::drawMouseDetectors()
{
    glInitNames();
    glPushName(0);

    //NOSE (openGL....z-axis)
    glLoadName(Z);
    glRotatef(90, 1, 0, 0);
    glRotatef(90, 0, 1, 0);
    z_axis->draw();

    //AZIMUTH (openGL....y-axis)
    glLoadName(Y);
    glRotatef(-90, 0, 1, 0);
    glRotatef(-90, 1, 0, 0);
    y_axis->draw();

    //ELEVATION (openGL....x-axis)
    glLoadName(X);
    glRotatef(90, 1, 0, 0);
    glRotatef(-90, 0, 0, 1);
    x_axis->draw();

    return;
}

```

```

***** : GlutBaseClass.h
// DESCRIPTION: Covers glut functions and is used for windowing tasks
*****


#ifndef __GLUTBASECLASS_H__
#define __GLUTBASECLASS_H__


#include <GL/glut.h>
#include <iostream.h>
#include "utility.h"
#include "Human.h"
#include "Floor.h"
#include "ProceduralAnim.h"
#include "UserControl.h"
#include "KeyFrameAnim.h"
#include "SensorSystem.h"


// ENUMS
enum VIEW_TYPE { WALK, PAN, STUDY };      // view mode types

enum POSTURE_CONTROL_TYPE { USER_CONTROL, WALKING, KEY_FRAME, SENSOR };

// CLASS DEFINITION
class GlutBaseClass {

public :-----P U B L I C-----


    //CONSTRUCTORS
    GlutBaseClass();                      //default
    GlutBaseClass( GlutBaseClass & );      //copy
    GlutBaseClass(GLint argc, char **argv ); //others


private :-----P R I V A T E-----


    //CONSTS
    static const short VIEWPOINT_Z;
    static const GLint WIN_POS_X,
                    WIN_POS_Y;

    //OPERATORS
    GlutBaseClass& operator=( const GlutBaseClass & );


    //FUNCTIONS
        //Event handling functions
    static void display();                  // set view and calls draw functions
    static void keyboard(unsigned char key, GLint x, GLint y); //handle keyboard

    static void activeMouseMotion(GLint x, GLint y);           //handle mouse motion

    static void visibility(GLint status);                  // set idle function for window visibility
    static void menuStatus(GLint status, GLint x, GLint y); //set idle function for

    static void reshape(GLint w, GLint h);                // change window settings for reshape
    static void animate();                                // makes animation, called by idle func

```

```

        // Menu Functions
static void mainMenu(GLint value);           // handle sub menus
static void viewSwitchMenu(GLint value);      // handle view (STUDY, WALK, PAN) , RESTORE
static void modelSwitchMenu(GLint value);     // handle drawing ( realistic, wireframe, stick )
static void UserControlMenu(GLint value);     // handle motion control
static void ProceduralAnimMenu(GLint value);
static void KeyFrameAnimMenu(GLint value);

        // viewing functions
static void setView(GLint x, GLint y);        // change view coord.(camera coord.)

        // Initialization Functions
void setDisplayMode();                      //sets display mode
void positionAndSizeWindows();              // sets initial coord. and size for window
void createDrawingData();                  // for Display List creation
void openGLInit();                        // initialize settings of world (light, drawing, ...)
void setUpMenus();                         // registerCallBacks();

//OBJECTs
static Human      humanObj; // objects that is controled, manupulated and drawn
static Floor       floor;
static ProceduralAnim procedure;
static UserControl userControl;
static KeyFrameAnim keyFrame;
static SensorSystem sensorSystem;

//VARIABLEs
static GLfloat
    viewPointX, viewPointY, viewPointZ,           // camera coord.
    rotationX, rotationY;                      // camera orientation
static GLint
    topMenu, menuView, menuModel, menuMotion,   // Menu identifiers
    menuUserControl, menuProceduralAnim, menuKeyFrameAnim,
    winSizeX, winSizeY,                         // Window dimensions
    oldMouseX, oldMouseY;                      // holds old mouse coord. before motion

static VIEW_TYPE
    viewPointMode;                            // view mode

static POSTURE_CONTROL_TYPE
    postureControlMode;

};

#endif

```

```

***** : GlutBaseClass.cpp
// DESCRIPTION: window specific functions
*****


#include "GlutBaseClass.h"

*****INITIALIZE STATIC DATA MEMBERS *****

Human      GlutBaseClass::humanObj;
Floor       GlutBaseClass::floor;
ProceduralAnim GlutBaseClass::procedure( &humanObj );
UserControl  GlutBaseClass::userControl( &humanObj );
KeyFrameAnim GlutBaseClass::keyFrame( &humanObj );
SensorSystem GlutBaseClass::sensorSystem( &humanObj );


const short GlutBaseClass::VIEWPOINT_Z =13; // initial view coord. for Z

const GLint GlutBaseClass::WIN_POS_X =50,
           GlutBaseClass::WIN_POS_Y =50;

GLint      GlutBaseClass::winSizeX(600),
           GlutBaseClass::winSizeY(600),
           GlutBaseClass::topMenu,
           GlutBaseClass::menuView,
           GlutBaseClass::menuModel,
           GlutBaseClass::menuMotion,
           GlutBaseClass::menuUserControl,
           GlutBaseClass::menuProceduralAnim,
           GlutBaseClass::menuKeyFrameAnim,
           GlutBaseClass::oldMouseX,
           GlutBaseClass::oldMouseY;

GLfloat GlutBaseClass::viewPointZ(VIEWPOINT_Z),
        GlutBaseClass::viewPointY(0),
        GlutBaseClass::viewPointX(0),
        GlutBaseClass::rotationX(0),
        GlutBaseClass::rotationY(0);

VIEW_TYPE GlutBaseClass::viewPointMode(STUDY);

POSTURE_CONTROL_TYPE GlutBaseClass::postureControlMode(USER_CONTROL);

*****END STATIC DATA MEMBER INITIALIZATION *****

//-----
GlutBaseClass::GlutBaseClass(GLint argc, char **argv)

{
    glutInit(&argc, argv);      // Initialize the GLUT library and negotiate
                                // a session with the window system.
    setDisplayMode();          // Set up INITIAL display mode

    positionAndSizeWindows(); // Set up the INITIAL windows

    registerCallBacks();       // Register INITIAL event handling functions
}

```

```

setUpMenus();           // Set up INITIAL Menus

openGLInit();          // Complete OpenGL rendering initialization

glutMainLoop();        // Enter the GLUT event processing loop.

}

//*****Initialization Functions *****

//-----
void GlutBaseClass::setDisplayMode()
{
    // Set initial display mode for double buffering and RGBA color
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB );
}

//-----
void GlutBaseClass::positionAndSizeWindows()
{
    // Specify window placement
    // Default placement is left to the window system
    glutInitWindowPosition(WIN_POS_X, WIN_POS_Y);

    // Specify window placement
    // Default window size is 300 x 300
    glutInitWindowSize(winSizeX, winSizeY);

    // Create a window entitled "Planets" and make it the current window
    glutCreateWindow("Body Motions");
}

//-----
void GlutBaseClass::openGLInit()
{
    //Polygon drawing
    glShadeModel (GL_SMOOTH);           // Explicitly set shade model to default
    glCullFace( GL_BACK );             // discard back faces
    glEnable( GL_CULL_FACE );

    //Depth test
    glClearDepth(1.0f);                // Specify value to clear the depth buffer
    glDepthFunc( GL_LEQUAL);           // Specify value used for depth compares
    glEnable( GL_DEPTH_TEST );          // Enable Depth tests

    //Material
    const GLfloat AMBIENT[3] ={ 0.6f, 0.430792f, 0.379119f },
                  SPECULAR[3]={ 0, 0, 0.5f };

    glMaterialfv(GL_FRONT, GL_AMBIENT, AMBIENT);
    glMaterialfv(GL_FRONT, GL_SPECULAR, SPECULAR);
    glMaterialf(GL_FRONT, GL_SHININESS, 5);

    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_DIFFUSE);
}

```

```

//Lights
//.....LIGHT 0.....
const GLfloat POSITION_0[4]={0,0,10,0};

glLightfv(GL_LIGHT0, GL_POSITION, POSITION_0);
glLightfv(GL_LIGHT0, GL_SPECULAR, SPECULAR);
glEnable(GL_LIGHT0);

glEnable(GL_LIGHTING);

createDrawingData();
}

//-----
void GlutBaseClass::createDrawingData()
{
    // Set Up Display Lists
}

//-----
void GlutBaseClass::registerCallBacks()
{
    glutDisplayFunc(display);           // callback for window redisplay
    glutKeyboardFunc(keyboard);        // callback for ascii character input
    glutSpecialFunc(specialKeys);      // callback for special keystrokes
    glutMouseFunc(mouseButton);        // callback for mouse button events
    glutMotionFunc(activeMouseMotion); // callback for mouse motion with the
                                      // buttons depressed.
    glutMenuStatusFunc(menuStatus);    // callback for menu exposures
    glutReshapeFunc(reshape);          // callback for window size changes
    glutVisibilityFunc(visibility);    // callback for visibility changes
    glutIdleFunc/animate();          // idle callback
}

//***** Menu Functions *****
//-----
void GlutBaseClass::setUpMenus()
{
    // Create viewPoint submenu.....
    menuView = glutCreateMenu(viewSwitchMenu);
    // Specify menu items and their GLint integer identifiers
    glutAddMenuEntry("Pan", 1);
    glutAddMenuEntry("Walk", 2);
    glutAddMenuEntry("Study", 3);
    glutAddMenuEntry("Restore view", 4);
}

```

```

// Create Model submenu.....  

menuModel = glutCreateMenu(modelSwitchMenu);  

    glutAddMenuEntry("Stick", 1);  

    glutAddMenuEntry("Skin", 2);  

    glutAddMenuEntry("Wire Frame", 3);  

// Create motion submenu.....  

menuUserControl = glutCreateMenu(UserControlMenu);  

    glutAddMenuEntry("Forward Kinematics", 1);  

    glutAddMenuEntry("Inverse Kinematic", 2);  

    glutAddMenuEntry("Restore posture", 3);  

menuProceduralAnim = glutCreateMenu(ProceduralAnimMenu);  

    glutAddMenuEntry("Walking( INV. KIN. )", 1);  

    glutAddMenuEntry("Walking( FWD. KIN. )", 2);  

menuKeyFrameAnim = glutCreateMenu(KeyFrameAnimMenu);  

    glutAddMenuEntry("(a)dd", 1);  

    glutAddMenuEntry("(i)nsert", 2);  

    glutAddMenuEntry("(d)elete", 3);  

    glutAddMenuEntry("(f)irst", 4);  

    glutAddMenuEntry("(l)ast", 5);  

    glutAddMenuEntry("(n)ext", 6);  

    glutAddMenuEntry("(p)revious", 7);  

    glutAddMenuEntry("(r)un", 8);  

    glutAddMenuEntry("(s)top", 9);  

menuMotion = glutCreateMenu(mainMenu);  

glutAddMenuEntry("InertialSensor", 1);  

glutAddSubMenu("User Control", menuUserControl);  

glutAddSubMenu("ProceduralAnim", menuProceduralAnim);  

glutAddSubMenu("KeyFrameAnim", menuKeyFrameAnim);  

// Create main menu.....  

topMenu = glutCreateMenu(mainMenu);  

    glutAddSubMenu("View Mode", menuView); // Attach view Menu  

    glutAddSubMenu("Model Mode", menuModel); // Attach model Menu  

    glutAddSubMenu("Posture Control", menuMotion); // Attach motion Menu  

    glutAttachMenu(GLUT_RIGHT_BUTTON); // Attach menu to right mouse button  

}  

//-----  

void GlutBaseClass::mainMenu(GLint value)  

{
    switch (value) {
    case(1):postureControlMode = SENSOR;
        break;
    default:
        cout << "Unknown Main Menu Selection!" << endl;
    }
}

```

```

//-----
void GlutBaseClass::viewSwitchMenu(GLint value)
{
    switch (value) {
    case(1):
        viewPointMode =PAN;
        break;
    case(2):
        viewPointMode =WALK;
        break;
    case(3):
        viewPointMode =STUDY;
        break;
    case(4):
        viewPointX =0;
        viewPointY =0;
        viewPointZ =VIEWPOINT_Z;
        rotationX =0;
        rotationY =0;
        break;
    default:
        //do nothing
    }
    // Signal GLUT to call display callback
    glutPostRedisplay();
}

//-----
void GlutBaseClass::modelSwitchMenu(GLint value)
{
    switch (value) {
    case(1):
        humanObj.setModelType( STICK );
        break;
    case(2):
        humanObj.setModelType( SKIN );
        break;
    case(3):
        humanObj.setModelType( WIRE_FRAME );
        break;
    default:
        //do nothing
    }
    // Signal GLUT to call display callback
    glutPostRedisplay();
}

//-----
void GlutBaseClass::UserControlMenu(GLint value)
{
    if( postureControlMode ==WALKING ){

        humanObj.initializePosture();
    }
    postureControlMode =USER_CONTROL;
}

```

```

switch (value) {
case(1):
    userControl.setControlType( QUATERNION_CONTROL );
    break;
case(2):
    userControl.setControlType( INVERSE_CONTROL );
    break;
case(3):
    humanObj.initializePosture();
    break;
default;;
    //do nothing
}

// Signal GLUT to call display callback
glutPostRedisplay();
}

//-----
void GlutBaseClass::ProceduralAnimMenu(GLint value)
{
    humanObj.initializePosture();

    userControl.setControlType( QUATERNION_CONTROL );
    postureControlMode =WALKING;

    switch (value) {
case(1):
    procedure.setWalkingMethod(INVERSE);
    break;
case(2):
    procedure.setWalkingMethod(FORWARD);
    break;
default;;
    //do nothing
}
// Signal GLUT to call display callback
glutPostRedisplay();
}

//-----
void GlutBaseClass::KeyFrameAnimMenu(GLint value)
{
    if( postureControlMode == USER_CONTROL ) {

        keyFrame.keyPressed( 'a' + value );

        if( value ==8 ){

            userControl.setControlType( QUATERNION_CONTROL );
            postureControlMode =KEY_FRAME;
        }
    }
}

```

```

// **** EVENT HANDLERS ****

//-----
void GlutBaseClass::display()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glLoadIdentity();           // Reset the modelview matrix

    if( postureControlMode == USER_CONTROL ) {

        userControl.mouseDragAt( oldMouseX, oldMouseY );
    }

    // Set the view point
    gluLookAt(viewPointX, viewPointY, viewPointZ,
              viewPointX, viewPointY, viewPointZ-VIEWPOINT_Z,
              0.0, 1.0, 0.0);

    glRotatef( rotationX, 1, 0, 0 );
    glRotatef( rotationY, 0, 1, 0 );

    floor.draw(); //draw the floor

    humanObj.draw(); //draw human obj

    glutSwapBuffers(); // Flush all drawing commands and swapbuffers
}

//-----
void GlutBaseClass::reshape(GLint w, GLint h)
{
    winSizeY = (h == 0) ? 1 : h;
    winSizeX = (w == 0) ? 1 : w;

    glViewport(0, 0, winSizeX, winSizeY);      // Set viewport to entire client area

    glMatrixMode (GL_PROJECTION);           // Projection matrix stack

    glLoadIdentity();                      // Reset the projection marix stack

    // Set up a perspective projection matrix
    gluPerspective(45.0f, (GLfloat)winSizeX/(GLfloat)winSizeY, 1.0f, 50.0f);

    glMatrixMode (GL_MODELVIEW);           // Modelview matrix stack
}

//-----
void GlutBaseClass::keyboard(unsigned char key, GLint x, GLint y)
{
    if( postureControlMode == USER_CONTROL ) {

        keyFrame.keyPressed( key );
    }
}

```

```

        if( key =='r' ){
            userControl.setControlType( QUATERNION_CONTROL );
            postureControlMode =KEY_FRAME;
        }
    }
    glutPostRedisplay();
}

//-----
void GlutBaseClass::specialKeys(GLint key, GLint x, GLint y)
{
    switch(key) {
        case(GLUT_KEY_UP):
            if( postureControlMode == WALKING ){
                procedure.increaseFrameRate();
                floor.increaseFrameRate();
            }
            break;
        case(GLUT_KEY_DOWN):
            if( postureControlMode == WALKING ){
                procedure.decreaseFrameRate();
                floor.decreaseFrameRate();
            }
            break;
        case(GLUT_KEY_RIGHT):
            break;
        case(GLUT_KEY_LEFT):
            break;
        default:
            break;
    }
}

//-----
void GlutBaseClass::mouseButton(GLint btn, GLint state, GLint x, GLint y)
{
    switch (btn) {
        case (0)://Left button
            if( postureControlMode == USER_CONTROL ) {

                (state == GLUT_DOWN ) ?
                    userControl.mouseHitAt( x, y, rotationX, rotationY )
                :
                    userControl.mouseReleasedAt( x, y );
            }
            break;
        case (1)://Center button
            break;
        case (2)://Right button
            break;
        default://Unknown button
            break;
    }
    oldMouseX =x; oldMouseY=y;
    glutPostRedisplay();
}

```

```

//-----
void GlutBaseClass::activeMouseMove(GLint x, GLint y)
{
    if ( userControl.isTracking() == FALSE ) {
        // if there is no selection on body parts,
        // this mouse motion is for view transformations
        setView(x,y);
    }

    // capture mouse coord
    oldMouseX =x; oldMouseY=y;

    glutPostRedisplay();
}

//-----
void GlutBaseClass::setView(GLint x, GLint y)
{
    static const GLfloat VIEW_STEP =0.5f; // step in world coord for WALK & PAN mode
    static const GLfloat ROTATION_STEP =20;// step of rotation on Y-axis for STUDY

    // oldMouseY chances affect viewPointY or viewPointZ or rotationX
    if(y-oldMouseY > 0){

        (viewPointMode == WALK) ? ( viewPointZ+=VIEW_STEP ) :
        ((viewPointMode == PAN) ? ( viewPointY-=VIEW_STEP ) :
            ( rotationX+=ROTATION_STEP ));

    }
    else if(y-oldMouseY < 0){

        (viewPointMode == WALK) ? ( viewPointZ-=VIEW_STEP ) :
        ((viewPointMode == PAN) ? ( viewPointY+=VIEW_STEP ) :
            ( rotationX-=ROTATION_STEP ));

    }

    // oldMouseX chances affect viewPointX or rotationY (depending on viewPointMode)
    if(x-oldMouseX > 0){

        (viewPointMode == STUDY) ? ( rotationY+=ROTATION_STEP ) :
            ( viewPointX+=VIEW_STEP );

    }
    else if(x-oldMouseX < 0){

        (viewPointMode == STUDY) ? ( rotationY-=ROTATION_STEP ) :
            ( viewPointX-=VIEW_STEP );

    }
}

//-----
void GlutBaseClass::visibility(GLint status)
{
    if (status == GLUT_VISIBLE) {
        glutIdleFunc(animate); // Turn on animation
    }
}

```

```

        else {
            glutIdleFunc(NULL);           // Turn off animation
        }
    }

//-----
void GlutBaseClass::menuStatus(GLint status, GLint x, GLint y)
{
    if (status == GLUT_MENU_IN_USE) {
        glutIdleFunc(NULL);           // Turn off animation
    }
    else {
        glutIdleFunc/animate);      // Turn on animation
    }
}

//-----
void GlutBaseClass::animate()
{
    if( postureControlMode == WALKING ){

        procedure.walk();
        floor.slide();
        display(); //draw again
    }
    else if( postureControlMode == KEY_FRAME ){

        if( !keyFrame.play() ){

            postureControlMode =USER_CONTROL;
        }
        display(); //draw again
    }
    else if( postureControlMode == SENSOR ){

        sensorSystem.trackSegment();
        display(); //draw again
    }
}

```

```

//***** : H_Matrix.h
// DESCRIPTION: general purpose homogeneous matrix
//*****  

#ifndef __H_MATRIX_H__
#define __H_MATRIX_H__  

#include <GL/glut.h>
#include "QuaternionR.h"  

class H_Matrix{  

public :-----P U B L I C-----  

//CONSTRUCTORS
H_Matrix();           //default
H_Matrix( H_Matrix & ); //copy  

//DESTRUCTOR
~H_Matrix();  

//FUNCTIONs
virtual void setBoundaries( const GLfloat* )=0;
void setTranslation( const GLfloat, const GLfloat, const GLfloat );
boolean setOrientation( const ROTATION_METHODS ,const GLfloat* );
void getOrientation( const ROTATION_METHODS , GLfloat * const );
boolean rotate( const ROTATION_METHODS, const GLfloat * );
void applyToCurrentMatrix();  

protected :-----P R O T E C T E D-----  

//OPERATORS
H_Matrix& operator=( const H_Matrix & );  

//FUNCTIONs
void quatToMatrix();
void vecAngToMatrix();
void matrixToQuat();
void quatToVecAng();
void vecAngToQuat();
boolean rotateByVecAng( const GLfloat * orientation );  

virtual void eulerToMatrix() =0;
virtual void matrixToEuler() =0;
virtual void setEuler( const GLfloat * )=0;
virtual void getEuler( GLfloat * const )=0;
virtual boolean rotateByEuler( const GLfloat * ) = 0;
virtual boolean isRotationAcceptable( QuaternionR & )=0;  

//OBJECTs
QuaternionR quaternion;  

//VARIABLEs
GLfloat angle, vector_x, vector_y, vector_z,
        matrix[16], *boundaries;
ROTATION_METHODS lastUpdateMethod;  

};  

#endif

```

```

//*****
// FILE : H_Matrix.cpp
// DESCRIPTION:
//*****

#include "H_Matrix.h"

//-----
H_Matrix::H_Matrix(){

    // INITIALIZE
    quatToMatrix();
    matrix[3] = 0 ;
    matrix[7] = 0 ;
    matrix[11]=0 ;
    matrix[15]=1 ;

    lastUpdateMethod =QUATERNION;
}

//-----
H_Matrix::~H_Matrix(){

}

//-----
void H_Matrix::applyToCurrentMatrix()
{
    glMultMatrixf( matrix );
}

//-----
void H_Matrix::setTranslation( const GLfloat XX,
                               const GLfloat YY,
                               const GLfloat ZZ )
{
    matrix[12]=XX ;
    matrix[13]=YY ;
    matrix[14]=ZZ ;

    return;
}

//-----
boolean H_Matrix::setOrientation( const ROTATION_METHODS method,
                                  const GLfloat* orientation)
{
    lastUpdateMethod =method;

    boolean rotationAccepted =FALSE;
}

```

```

switch( method ){

    case VECTOR_ANGLE:{

        QuaternionR newOrientation( orientation );
        rotationAccepted =isRotationAcceptable( newOrientation );
        if ( rotationAccepted ){

            angle =orientation[3];
            vector_x =orientation[X];
            vector_y =orientation[Y];
            vector_z =orientation[Z];
            vecAngToMatrix();
        }
        }break;
    case QUATERNION:{

        QuaternionR newOrientation;
        newOrientation.setValues( orientation );

        rotationAccepted =isRotationAcceptable( newOrientation );
        if ( rotationAccepted ){

            quaternion= newOrientation;
            quatToMatrix();
        }
        }break;
    case EULER:

        rotationAccepted =rotateByEuler( orientation );
        break;
    case MATRIX:{

        matrixToQuat();
        QuaternionR old =quaternion;
        matrix[0] =orientation[0]; matrix[4] =orientation[3]; matrix[8] =orientation[6];
        matrix[1] =orientation[1]; matrix[5] =orientation[4]; matrix[9] =orientation[7];
        matrix[2] =orientation[2]; matrix[6] =orientation[5]; matrix[10] =orientation[8];
        matrixToQuat();
        rotationAccepted =isRotationAcceptable( quaternion );
        if( !rotationAccepted ){

            quaternion =old;
            quatToMatrix();
        }
        }break;
    }

    if( rotationAccepted ) lastUpdateMethod =method;

    return rotationAccepted;
}

```

```

//-----
void H_Matrix::getOrientation( const ROTATION_METHODS method,
                               GLfloat* const orientation)
{
    switch( method ){
        case VECTOR_ANGLE:
            if( lastUpdateMethod !=VECTOR_ANGLE ) matrixToQuat();
            quatToVecAng();
            orientation[3] =angle;
            orientation[X] =vector_x;
            orientation[Y] =vector_y;
            orientation[Z] =vector_z;
            break;
        case QUATERNION:
            if( lastUpdateMethod ==VECTOR_ANGLE ) vecAngToQuat();
            else if( lastUpdateMethod !=QUATERNION ) matrixToQuat();

            quaternion.getValues( orientation );
            break;
        case EULER:
            if( lastUpdateMethod !=EULER ) matrixToEuler();
            getEuler( orientation );
            break;
        case MATRIX://never called
            break;
    }
}

//-----
boolean H_Matrix::rotate( const ROTATION_METHODS method,
                          const GLfloat* orientation)
{
    boolean rotationAccepted =FALSE;

    switch( method ){
        case VECTOR_ANGLE:
            if( lastUpdateMethod !=VECTOR_ANGLE ) matrixToQuat();

            rotationAccepted =rotateByVecAng( orientation );

            if( rotationAccepted ) lastUpdateMethod =QUATERNION;

            break;
        case QUATERNION://never called
            break;
        case EULER:
            rotationAccepted =rotateByEuler( orientation );

            if( rotationAccepted ) lastUpdateMethod =EULER;

            break;
    }
}

```

```

        case MATRIX://never called
            break;
    }

    return rotationAccepted;
}

//-----
boolean H_Matrix::rotateByVecAng( const GLfloat * orientation )
{
    boolean rotationAccepted =FALSE;

    QuaternionR rotation( orientation );

    // apply rotation on existing quaternion orientation
    QuaternionR newOrientation =rotation * quaternion;

    //checks if rotation in boundaries
    if( isRotationAcceptable( newOrientation ) ){

        quaternion =newOrientation;

        quatToMatrix();

        rotationAccepted =TRUE;// if it is, return accepted
    }

    return rotationAccepted;
}

//-----
void H_Matrix::quatToVecAng()
{
    GLfloat vectorScalar,
            quat[THREE_D+1];

    quaternion.getValues( quat );

    angle =acos(quat[3]) * 2;
    vectorScalar =sin( angle/2 );
    if( vectorScalar != 0 ){
        vector_x =quat[X] / vectorScalar;
        vector_y =quat[Y] / vectorScalar;
        vector_z =quat[Z] / vectorScalar;
    }
    else{
        vector_x =vector_y =vector_z =0;
    }
    angle =angle / DEG_TO_RAD;
}

```

```

//-----
void H_Matrix::vecAngToQuat()
{
    quaternion =QuaternionR( vector_x ,vector_y ,vector_z, angle );
    quaternion.normalize();

    return;
}

//-----
void H_Matrix::matrixToQuat()
{
    GLfloat quat[THREE_D+1];

    quat[3] =(GLfloat) sqrt( (matrix[0]+matrix[5]+matrix[10]+1)/4 );

    GLfloat w4 =4*quat[3];

    quat[X] =(matrix[6]-matrix[9]) / w4;
    quat[Y] =(matrix[8]-matrix[2]) / w4;
    quat[Z] =(matrix[1]-matrix[4]) / w4;

    quaternion.setValues( quat );
}

//-----
void H_Matrix::quatToMatrix(){

    GLfloat quat[THREE_D];

    quaternion.getValues( quat );

    GLfloat
        xx =quat[X]*quat[X], yy =quat[Y]*quat[Y], zz =quat[Z]*quat[Z],
        xy =quat[X]*quat[Y], xz =quat[X]*quat[Z], yz =quat[Y]*quat[Z],
        wx =quat[X]*quat[3], wy =quat[Y]*quat[3], wz =quat[Z]*quat[3];

    matrix[0] = 1 - (2* (yy+zz)) ;
    matrix[1] = 2* (xy + wz);
    matrix[2] = 2* (xz - wy);

    matrix[4] = 2* (xy - wz) ;
    matrix[5] = 1 - (2* (xx+zz)) ;
    matrix[6] = 2* (yz + wx) ;

    matrix[8] = 2* (xz + wy) ;
    matrix[9] = 2* (yz - wx) ;
    matrix[10] = 1 - (2* (xx+yy)) ;

    return;
}

```

```

//-----
void H_Matrix::vecAngToMatrix()
{
    GLfloat
    s  =(GLfloat) sin( angle*DEG_TO_RAD ),
    c  =(GLfloat) cos( angle*DEG_TO_RAD ),
    ci =1 - c,
    xyci =vector_x*vector_y*ci,
    xzci =vector_x*vector_z*ci,
    yzci =vector_y*vector_z*ci,
    xs =vector_x*s , ys =vector_y*s , zs =vector_z*s;

    matrix[0] = (vector_x*vector_x*ci) + c ;
    matrix[1] = xyci +zs;
    matrix[2] = xzci -ys;

    matrix[4] = xyci -zs ;
    matrix[5] = (vector_y*vector_y*ci) + c ;
    matrix[6] = yzci + xs ;

    matrix[8] = xzci + ys ;
    matrix[9] = yzci - xs ;
    matrix[10] = (vector_z*vector_z*ci) + c ;
}

```

```

//***** : H_Matrix1DOF.h
// DESCRIPTION: special functions for 1 DOF
//***** 

#ifndef __H_MATRIX1DOF_H__
#define __H_MATRIX1DOF_H__

#include <GL/glut.h>
#include "H_Matrix.h"

class H_Matrix1DOF: public H_Matrix{

public ://-----P U B L I C-----

    //CONSTRUCTORS
    H_Matrix1DOF( AXIS );           //default
    H_Matrix1DOF( H_Matrix1DOF & ); //copy

    //DESTRUCTOR
    ~H_Matrix1DOF();

    //FUNCTIONS
    void setBoundaries( const GLfloat* );
    boolean rotateByEuler( const GLfloat * );
    void setEuler( const GLfloat * );
    void getEuler( GLfloat * const );

private ://-----P R I V A T E-----

    //OPERATORS
    H_Matrix1DOF& operator=( const H_Matrix1DOF & );

    //FUNCTIONS
    void matrixToEuler();
    void eulerToMatrix();
    void eulerXToMatrix( GLfloat, GLfloat );
    void eulerYToMatrix( GLfloat, GLfloat );
    void eulerZToMatrix( GLfloat, GLfloat );

    boolean isRotationAcceptable( QuaternionR & );
    boolean isRotationAcceptable( GLfloat );
    boolean isMatrixFit();

    //VARIABLES
    GLfloat
        angle, minAngle, maxAngle;

    AXIS axis;
};

#endif

```

```

//*****
// FILE : H_Matrix1DOF.cpp
// DESCRIPTION:
//*****
#include "H_Matrix1DOF.h"

//-----
H_Matrix1DOF::H_Matrix1DOF( AXIS a){

    // INITIALIZE
    axis =a;
    angle =0;
}

//-----
H_Matrix1DOF::~H_Matrix1DOF(){

}

//-----
void H_Matrix1DOF::setBoundaries( const GLfloat* boundaries )
{
    minAngle =boundaries[ 2*axis ];
    maxAngle =boundaries[ (2*axis)+1 ];
}

//-----
void H_Matrix1DOF::setEuler( const GLfloat * eulerAngle )
{
    switch( axis ){
        case X:angle =eulerAngle[X];
            break;
        case Y:angle =eulerAngle[Y];
            break;
        case Z:angle =eulerAngle[Z];
            break;
    }

    eulerToMatrix();
}

//-----
void H_Matrix1DOF::getEuler( GLfloat * const eulerAngle )
{
    eulerAngle[X] =OUT_RANGE;
    eulerAngle[Y] =OUT_RANGE;
    eulerAngle[Z] =OUT_RANGE;

    eulerAngle[axis] =angle;
}

//-----
boolean H_Matrix1DOF::rotateByEuler( const GLfloat *eulerAngle )
{
    boolean rotationAccepted =FALSE;
}

```

```

//checks if rotation in boundaries
if( isRotationAcceptable( eulerAngle[axis] ) ){

    angle =eulerAngle[axis];

    eulerToMatrix();

    rotationAccepted =TRUE;// if it is, return accepted
}

return rotationAccepted;
}

//-----
boolean H_Matrix1DOF::isRotationAcceptable( QuaternionR & newOrientation )
{
    boolean accepted =FALSE; //default, don't accept rotation

    H_Matrix1DOF tmp( axis );

    tmp.minAngle =minAngle;
    tmp.maxAngle =maxAngle;

    tmp.quaternion = newOrientation;

    tmp.quatToMatrix();

    tmp.matrixToEuler();

    if( tmp.isRotationAcceptable( tmp.angle ) && tmp.isMatrixFit() ){

        accepted =TRUE;
    }

    return accepted;
}

//-----
boolean H_Matrix1DOF::isRotationAcceptable( GLfloat a )
{
    if( a > 180 ) a =-360;

    if( a <minAngle || a >maxAngle ){

        return FALSE;
    }

    return TRUE;
}

//-----
boolean H_Matrix1DOF::isMatrixFit()
{
    boolean accepted =FALSE; //default, don't accept rotation
}

```

```

if(
    ( axis ==X && matrix[0] ==1 && matrix[1] ==0 &&
        matrix[2] ==0 && matrix[4] ==0 && matrix[8] ==0)
    ||
    ( axis ==Y && matrix[1] ==0 && matrix[4] ==0 &&
        matrix[5] ==1 && matrix[6] ==0 && matrix[9] ==0)
    ||
    ( axis ==Z && matrix[2] ==0 && matrix[6] ==0 &&
        matrix[8] ==0 && matrix[9] ==0 && matrix[10] ==1)
){
    accepted =TRUE;
}
return accepted;
}

//-----
void H_Matrix1DOF::matrixToEuler()
{
    switch( axis ){
        case X:angle =(GLfloat)acos(matrix[5]) / DEG_TO_RAD;
            if( matrix[6] < 0 ) angle =360-angle;
            break;
        case Y:angle =(GLfloat)acos(matrix[0]) / DEG_TO_RAD;
            if( matrix[8] < 0 ) angle =360-angle;
            break;
        case Z:angle =(GLfloat)acos(matrix[0]) / DEG_TO_RAD;
            if( matrix[1] < 0 ) angle =360-angle;
            break;
    }
}

//-----
void H_Matrix1DOF::eulerToMatrix()
{
    GLfloat
        ca =(GLfloat) cos( angle * DEG_TO_RAD ),
        sa =(GLfloat) sin( angle * DEG_TO_RAD );

    switch( axis ){
        case X:eulerXToMatrix( ca, sa );
            break;
        case Y:eulerYToMatrix( ca, sa );
            break;
        case Z:eulerZToMatrix( ca, sa );
            break;
    }
}

//-----
void H_Matrix1DOF::eulerXToMatrix( const GLfloat cosAngle,
                                    const GLfloat sinAngle )
{
    matrix[0] =1;
    matrix[1] =0;
    matrix[2] =0;
}

```

```

matrix[4] = 0;
matrix[5] = cosAngle;
matrix[6] = sinAngle;

matrix[8] = 0;
matrix[9] = -sinAngle;
matrix[10] = cosAngle;
}

//-----
void H_Matrix1DOF::eulerYToMatrix( const GLfloat cosAngle,
                                   const GLfloat sinAngle )
{
    matrix[0] = cosAngle;
    matrix[1] = 0;
    matrix[2] = -sinAngle;

    matrix[4] = 0;
    matrix[5] = 1;
    matrix[6] = 0;

    matrix[8] = sinAngle;
    matrix[9] = 0;
    matrix[10] = cosAngle;
}

//-----
void H_Matrix1DOF::eulerZToMatrix( const GLfloat cosAngle,
                                   const GLfloat sinAngle )
{
    matrix[0] = cosAngle;
    matrix[1] = sinAngle;
    matrix[2] = 0;

    matrix[4] = -sinAngle;
    matrix[5] = cosAngle;
    matrix[6] = 0;

    matrix[8] = 0;
    matrix[9] = 0;
    matrix[10] = 1;
}

```

```

//*****
// FILE    : H_Matrix3DOF.h
// DESCRIPTION: Special functions for 3 DOF
//*****
#ifndef __H_MATRIX3DOF_H__
#define __H_MATRIX3DOF_H__

#include <GL/glut.h>
#include "H_Matrix.h"

class H_Matrix3DOF: public H_Matrix{

public ://-----P U B L I C-----

    //CONSTRUCTORS
    H_Matrix3DOF( );                                //default
    H_Matrix3DOF( H_Matrix3DOF & );    //copy

    //DESTRUCTOR
    ~H_Matrix3DOF();

    //FUNCTIONS
    void setBoundaries( const GLfloat* );
    boolean rotateByEuler( const GLfloat * );
    void setEuler( const GLfloat * );
    void getEuler( GLfloat * const );

private ://-----P R I V A T E-----

    //OPERATORS
    H_Matrix3DOF& operator=( const H_Matrix3DOF & );

    //FUNCTIONS
    void matrixToEuler();
    void eulerToMatrix();

    boolean isRotationAcceptable( QuaternionR & );
    boolean isRotationAcceptable( const GLfloat * );

    //VARIABLES
    GLfloat
        angle[THREE_D],
        minAngle[THREE_D], maxAngle[THREE_D];
};

#endif

```

```

//*****
// FILE    : H_Matrix3DOF.cpp
// DESCRIPTION:
//*****
#include "H_Matrix3DOF.h"

//-----
H_Matrix3DOF::H_Matrix3DOF{

    // INITIALIZE
    angle[X] =angle[Y] =angle[Z] =0;
}

//-----
H_Matrix3DOF::~H_Matrix3DOF{

}

//-----
void H_Matrix3DOF::setBoundaries( const GLfloat* boundaries )
{
    for( GLint i=X; i<=Z; i++ ){

        minAngle[i] =boundaries[ 2*i ];
        maxAngle[i] =boundaries[ (2*i)+1 ];
    }
}

//-----
boolean H_Matrix3DOF::rotateByEuler( const GLfloat *eulerAngle )
{
    boolean rotationAccepted =FALSE;

    //checks if rotation in boundaries
    if( isRotationAcceptable( eulerAngle ) ){

        setEuler( eulerAngle );

        rotationAccepted =TRUE;// if it is, return accepted
    }
    return rotationAccepted;
}

//-----
void H_Matrix3DOF::setEuler( const GLfloat * eulerAngle )
{
    for( GLint i=X; i<=Z; i++ ){

        angle[i] =eulerAngle[i];
    }

    eulerToMatrix();
}

```

```

//-----
void H_Matrix3DOF::getEuler( GLfloat * const eulerAngle )
{
    for( GLint i=X; i<=Z; i++ ){

        eulerAngle[i] =angle[i];
    }
}

//-----
boolean H_Matrix3DOF::isRotationAcceptable( QuaternionR & newOrientation )
{
    boolean accepted =FALSE; //default, don't accept rotation

    H_Matrix3DOF tmp;

    for( GLint i=X; i<=Z; i++ ){

        tmp.minAngle[i] =minAngle[i];
        tmp.maxAngle[i] =maxAngle[i];
    }

    tmp.quaternion = newOrientation;

    tmp.quatToMatrix();

    tmp.matrixToEuler();

    if( tmp.isRotationAcceptable( tmp.angle ) ){

        accepted =TRUE;
    }
}

return accepted;
}

//-----
boolean H_Matrix3DOF::isRotationAcceptable( const GLfloat *eulerAngle )
{
    for( GLint i=X; i<=Z; i++ ){

        GLfloat a =eulerAngle[i];
        if( a > 180 ) a -=360;

        if( a <minAngle[i] || a >maxAngle[i] ){

            return FALSE;
        }
    }

    return TRUE;
}

```

```

//-----
void H_Matrix3DOF::matrixToEuler()
{
    angle[Y] =(GLfloat) asin( -matrix[2] );

    GLfloat
        cy =(GLfloat) cos( angle[Y] ),

        cz =matrix[0] /cy,
        sz =matrix[1] /cy,
        sx =matrix[6] /cy,
        cx =matrix[10]/cy;

    angle[Z] = acos(cz) / DEG_TO_RAD;
    if( sz < 0 ) angle[Z] =360-angle[Z];

    angle[X] = acos(cx) / DEG_TO_RAD;
    if( sx < 0 ) angle[X] =360-angle[X];

    angle[Y] /=DEG_TO_RAD;
    if( angle[Y] < 0 ) angle[Y] +=360;
}

//-----
void H_Matrix3DOF::eulerToMatrix()
{
    GLfloat
        cx =(GLfloat) cos( angle[X] * DEG_TO_RAD ),
        cy =(GLfloat) cos( angle[Y] * DEG_TO_RAD ),
        cz =(GLfloat) cos( angle[Z] * DEG_TO_RAD ),
        sx =(GLfloat) sin( angle[X] * DEG_TO_RAD ),
        sy =(GLfloat) sin( angle[Y] * DEG_TO_RAD ),
        sz =(GLfloat) sin( angle[Z] * DEG_TO_RAD ),

        cxz =cx*cz, sxz =sx*sz, cszx =cz*sx, csxz =cx*sz;

    matrix[0] = cz*cy;
    matrix[1] = sz*cy;
    matrix[2] = -sy;

    matrix[4] = ( cszx * sy ) - csxz;
    matrix[5] = ( sxz * sy ) + cxz;
    matrix[6] = sx*cy;

    matrix[8] = ( cxz * sy ) + sxz;
    matrix[9] = ( csxz * sy ) - cszx;
    matrix[10] = cx*cy;
}

```

```

***** : Human.h
// DESCRIPTION: Composition of Segment objects
*****


#ifndef __HUMAN_H__
#define __HUMAN_H__


#include <GL/glut.h>
#include <iostream.h>
#include "Segment.h"
#include "Joint.h"


// ENUMs
enum SEGMENTS { ROOT, BODY, NECK, HEAD,           //names of human body segments
                L_HIP, L_LEG ,L_FOOT,
                L_UPPER_ARM, L_FORE_ARM, L_HAND,
                R_HIP, R_LEG ,R_FOOT,
                R_UPPER_ARM, R_FORE_ARM, R_HAND,
                NONE };

enum JOINTS { WAIST, BODY_NECK, NECK_HEAD,           //names of human body joints
              L_HIP_JOINT, L_KNEE ,L_ANKLE,
              L_SHOULDER, L_ELBOW, L_WRIST,
              R_HIP_JOINT, R_KNEE ,R_ANKLE,
              R_SHOULDER, R_ELBOW, R_WRIST };



//CLASS DEF.
class Human{

public :-----PUBLIC-----


    //CONSTRUCTORS
    Human();
    Human( Human &);

    //DESTRUCTOR
    ~Human();

    //FUNCTIONs
    void draw();
    void drawMouseDetectors();

    void initializePosture();
    void setModelType( MODEL_TYPE );

    boolean setPosture( const ROTATION_METHODS, const SEGMENTS, const GLfloat * );
    void getPosture( const ROTATION_METHODS, const SEGMENTS, GLfloat * const);
    boolean modifyPosture( const ROTATION_METHODS, const SEGMENTS, const GLfloat * );

    void  getJointCenters( const SEGMENTS ,GLfloat &, GLfloat &, GLfloat & );
    void  getEarthMatrix( SEGMENTS, GLfloat[16] );
    GLfloat getSegmentLength( SEGMENTS );
}

```

```

private ://-----P R I V A T E-----


//OPERATORS
Human& operator=( const Human & );


//FUNCTIONs
void constructHumanModel();
void constructJoints();
void constructSegmentShapes();

void drawRoot(); // draw functions for segments
void drawBody();
void drawHead();
void drawLeftArm();
void drawRightArm();
void drawLeftLeg();
void drawRightLeg();
void transformJoints( SEGMENTS ); //calls humanJoint objects for transform of joint points


//OBJECT POINTERS
Segment *segment[NUMBER_OF_SEGMENTS];
Joint *joint[NUMBER_OF_JOINTS];


//VARIABLEs
GLint
    numberOfPoints; // number of points that are used to draw human
GLfloat
    **points, //points to draw human
    **normals; //normals of all points

MODEL_TYPE modelType;

};

#endif

```

```

//*****
// FILE : Human.cpp
// DESCRIPTION:
//*****

#include "Human.h"

//-----
Human::Human()
{
    //INITIALIZE
    normals =points =NULL;
    modelType =STICK;

    for(GLint segmentNo=0; segmentNo< NUMBER_OF_SEGMENTS; segmentNo++){

        AXIS dof =XYZ;
        if( segmentNo ==L_LEG || segmentNo ==R_LEG ){ dof =X; }
        else if( segmentNo ==L_FORE_ARM || segmentNo ==R_FORE_ARM ){ dof =Z; }

        segment[ segmentNo ] =new Segment( segmentNo, dof );
    }

    for(GLint jointNo=0; jointNo< NUMBER_OF_JOINTS; jointNo++){

        joint[ jointNo ] =NULL;
    }

    segment[HEAD] ->setStickShape( Box( 1, 0.25f, -0.25f, 0.25f, -0.25f ) );

    Box hand( -0.7f, 0.08f, -0.08f, 0.2f, -0.2f );
    segment[L_HAND]->setStickShape( hand );
    segment[R_HAND]->setStickShape( hand );

    Box foot( -0.25f, 0.2f, -0.2f, 0.8f, -0.1f );
    segment[L_FOOT]->setStickShape( foot );
    segment[R_FOOT]->setStickShape( foot );
}

//-----
Human::~Human()
{
    for(int i=0; i<numberOfPoints; i++){

        delete [] points[i];
        delete [] normals[i];
    }

    delete [] points;
    delete [] normals;

    for(GLint segmentNo=0; segmentNo< NUMBER_OF_SEGMENTS; segmentNo++){

        delete segment[ segmentNo ];
    }
}

```

```

        for(GLint jointNo=0; jointNo< NUMBER_OF_JOINTS; jointNo++){
            delete joint[ jointNo ];
        }
    }

//-----
void Human::initializePosture()
{
    GLfloat resetOrientation[4] ={0,0,0,1};

    for(int segmentNo=0; segmentNo< NUMBER_OF_SEGMENTS; segmentNo++){
        segment[segmentNo] ->setOrientation( QUATERNION, resetOrientation );
    }

    return;
}

//-----
boolean Human::setPosture( const ROTATION_METHODS method,
                           const SEGMENTS SEGMENT,
                           const GLfloat * orientation )
{
    return segment[ SEGMENT ] ->setOrientation( method, orientation );
}

//-----
void Human::getPosture( const ROTATION_METHODS method,
                        const SEGMENTS SEGMENT,
                        GLfloat * const orientation )
{
    segment[SEGMENT] ->getOrientation( method, orientation );
}

//-----
boolean Human::modifyPosture( const ROTATION_METHODS method,
                               const SEGMENTS segmentNo,
                               const GLfloat *orientation )
{
    return segment[ segmentNo ] ->rotate( method, orientation );
}

//-----
void Human::drawMouseDetectors()
{
    glInitNames();
    glPushName(0);

    for(GLuint i=0; i<NUMBER_OF_SEGMENTS; i++){
        glLoadName(i);
        segment[i] ->drawDetectionVolume();
    }
    return;
}

```

```

***** DRAWING FUNC. *****

//-----
void Human::draw()
{
    // Root
    drawRoot();
    glPushMatrix();
        // Body
        drawBody();
        glPushMatrix();
            //Head
            drawHead();
        glPopMatrix();
        glPushMatrix();
            //LeftArm
            drawLeftArm();
        glPopMatrix();
        glPushMatrix();
            //RightArm
            drawRightArm();
        glPopMatrix();
        glPopMatrix();
        glPushMatrix();
            //LeftLeg
            drawLeftLeg();
        glPopMatrix();
        glPushMatrix();
            //RightArm
            drawRightLeg();
        glPopMatrix();

    return;
}

//-----
void Human::drawRoot()
{
    glColor3f(0, 0.01f, 0.8f);

    // Root
    transformJoints(ROOT);
    segment[ROOT] ->draw(points,normals);
    return;
}

//-----
void Human::drawBody(){

    // Body
    transformJoints(BODY);
    segment[BODY]->draw(points,normals);

    return;
}

```

```

//-----
void Human::drawHead(){

    glColor3f(0.6f, 0.430792f, 0.379119f);

    // Neck
    transformJoints(NECK);
    segment[NECK]->draw(points, normals);

    // Head
    transformJoints(HEAD);
    segment[HEAD]->draw(points, normals);

    return;
}

//-----
void Human::drawLeftArm(){

    glColor3f(0, 0.01f, 0.8f);

    // Right Upper Arm
    transformJoints(L_UPPER_ARM);
    segment[L_UPPER_ARM]->draw(points, normals);

    // Right Fore Arm
    transformJoints(L_FORE_ARM);
    segment[L_FORE_ARM]->draw(points, normals);

    glColor3f(0.6f, 0.430792f, 0.379119f);

    // Right Hand
    transformJoints(L_HAND);
    segment[L_HAND]->draw(points, normals);
    return;
}

//-----
void Human::drawRightArm(){

    glColor3f(0, 0.01f, 0.8f);
    // Right Upper Arm
    transformJoints(R_UPPER_ARM);
    segment[R_UPPER_ARM]->draw(points, normals);

    // Right Fore Arm
    transformJoints(R_FORE_ARM);
    segment[R_FORE_ARM]->draw(points, normals);

    glColor3f(0.6f, 0.430792f, 0.379119f);
    // Right Hand
    transformJoints(R_HAND);
    segment[R_HAND]->draw(points, normals);
    return;
}

```

```

//-----
void Human::drawLeftLeg(){

    glColor3f(0, 0.01f, 0.8f);

    // Left Hip
    transformJoints(L_HIP);
    segment[L_HIP]->draw(points, normals);

    // Left Leg
    transformJoints(L_LEG);
    segment[L_LEG]->draw(points, normals);

    glColor3f(0.6f, 0.430792f, 0.379119f);

    // Left Foot
    transformJoints(L_FOOT);
    segment[L_FOOT]->draw(points, normals);
    return;
}

//-----
void Human::drawRightLeg(){

    glColor3f(0, 0.01f, 0.8f);

    // Left Hip
    transformJoints(R_HIP);
    segment[R_HIP]->draw(points, normals);

    // Left Leg
    transformJoints(R_LEG);
    segment[R_LEG]->draw(points, normals);

    glColor3f(0.6f, 0.430792f, 0.379119f);

    // Left Foot
    transformJoints(R_FOOT);
    segment[R_FOOT]->draw(points, normals);

    return;
}

*****END DRAWING FUNC *****

//-----
void Human::transformJoints(SEGMENTS segmentName){

    if( modelType ==SKIN || modelType ==WIRE_FRAME ){

        switch( segmentName ){

            case ROOT :
                joint[WAIST]->transformToEnd(points);
                joint[R_HIP_JOINT]->transformToEnd(points);

```

```

        joint[L_HIP_JOINT]->transformToEnd(points);
        break;
    case BODY :
        joint[WAIST]->transformToHead(points);
        joint[L_SHOULDER]->transformToEnd(points);
        joint[R_SHOULDER]->transformToEnd(points);
        joint[BODY_NECK]->transformToEnd(points);
        break;
    case NECK :
        joint[BODY_NECK]->transformToHead(points);
        joint[NECK_HEAD]->transformToEnd(points);
        break;
    case HEAD :
        joint[NECK_HEAD]->transformToHead(points);
        break;
    case L_UPPER_ARM :
        joint[L_SHOULDER]->transformToHead(points);
        joint[L_ELBOW]->transformToEnd(points);
        break;
    case L_FORE_ARM :
        joint[L_ELBOW]->transformToHead(points);
        joint[L_WRIST]->transformToEnd(points);
        break;
    case L_HAND :
        joint[L_WRIST]->transformToHead(points);
        break;
    case R_UPPER_ARM :
        joint[R_SHOULDER]->transformToHead(points);
        joint[R_ELBOW]->transformToEnd(points);
        break;
    case R_FORE_ARM :
        joint[R_ELBOW]->transformToHead(points);
        joint[R_WRIST]->transformToEnd(points);
        break;
    case R_HAND :
        joint[R_WRIST]->transformToHead(points);
        break;
    case L_HIP :
        joint[L_HIP_JOINT]->transformToHead(points);
        joint[L_KNEE]->transformToEnd(points);
        break;
    case L_LEG :
        joint[L_KNEE]->transformToHead(points);
        joint[L_ANKLE]->transformToEnd(points);
        break;
    case L FOOT :
        joint[L_ANKLE]->transformToHead(points);
        break;
    case R_HIP :
        joint[R_HIP_JOINT]->transformToHead(points);
        joint[R_KNEE]->transformToEnd(points);
        break;
    case R_LEG :
        joint[R_KNEE]->transformToHead(points);
        joint[R_ANKLE]->transformToEnd(points);
        break;

```

```

        case R FOOT :
            joint[R ANKLE]->transformToHead(points);
            break;
        default // do nothing
            break;
    }// end of switch()

}// end of if

return;
}

//-----
void Human::constructHumanModel()
{
    cout << "Reading points..." << endl;
    points =readPoints("points.dat", numberOfPoints );

    cout << "Reading normal vectors..." << endl;
    normals =readPoints("normals.dat", numberOfPoints );

    if( points !=NULL && normals!=NULL ){

        // construct datas of joint objects
        constructJoints();

        modelType =SKIN;

        // construct datas of segment objects
        constructSegmentShapes();

    }
    else{
        cout << "Program terminated abnormally." << endl;
        exit(0);
    }
    cout << endl << "Human body is constructed." << endl;
}

//-----
void Human::constructJoints()
{
    cout << "Waiting for JOINTs construction..." << endl;

    for(GLint jointNo=0; jointNo< NUMBER_OF_JOINTS; jointNo++){

        joint[jointNo] =new Joint( jointNo, points );

        if( joint[jointNo]->isConstructed() ==FALSE ){

            cout << "Joints can't be constructed.Program terminated." << endl;
            exit(0);
        }
    }
}

```

```

//-----
void Human::constructSegmentShapes()
{
    cout << "Waiting for SEGMENTs construction..." << endl;

    for(GLint segmentNo=0; segmentNo< NUMBER_OF_SEGMENTS; segmentNo++){

        Joint * jointPoint =NULL;
        if( segmentNo!= ROOT ) jointPoint =joint[ segmentNo -1 ];
        transformJoints( (SEGMENTS) segmentNo );
        if( segment[segmentNo] ->constructShape( points, jointPoint ) ==FALSE ){

            cout << "Segments can't be constructed.Program terminated." << endl;
            exit(0);
        }
    }
}

//-----
void Human::setModelType( MODEL_TYPE type )
{
    if( points ==NULL ) constructHumanModel();

    for( GLint segmentNo=0; segmentNo< NUMBER_OF_SEGMENTS; segmentNo++){

        segment[segmentNo] ->setModelType( type );
    }

    modelType =type;

    return;
}

//-----
void Human::getEarthMatrix( SEGMENTS segmentNo, GLfloat h_matrix[16] )
{
    segment[segmentNo] ->getEarthMatrix( h_matrix );
}

//-----
GLfloat Human::getSegmentLength( SEGMENTS segmentNo )
{
    return segment[segmentNo] ->getLength();
}

//-----
void Human::getJointCenters( const SEGMENTS SEGMENT,
                            GLfloat & jointX, GLfloat &jointY, GLfloat &jointZ )
{
    segment[SEGMENT] ->getJointCenters( jointX, jointY, jointZ );
}

```

```

***** : InverseKinematics.h
// DESCRIPTION: functions for inv. kinematics of end-effectors
*****


#ifndef __INVERSEKINEMATICS_H__
#define __INVERSEKINEMATICS_H__


#include <GL/glut.h>
#include "utility.h"
#include "Human.h"

class InverseKinematics{

public :-----P U B L I C-----


    //CONSTRUCTORS
    InverseKinematics( Human *); //default
    InverseKinematics( InverseKinematics &); //copy

    //DESTRUCTOR
    ~InverseKinematics();


    //FUNCTIONs
    void initialize( SEGMENTS );
    void setEarthOrientation(const GLfloat rotX, const GLfloat rotY);
    void algebraicSolution(const GLfloat x, const GLfloat y, const GLfloat z);

private :-----P R I V A T E-----


    //OPERATORs
    InverseKinematics& operator=( const InverseKinematics &);

    //FUNCTIONs
    void calculateAllAnglesForArm();
    void calculateAllAnglesForLeg();
    void calculateEndEffectorH_Matrix(const GLfloat x, const GLfloat y, const GLfloat z);
    void calculateSegmentLengths();
    void eulerToMatrix_YXZ( GLdouble, GLdouble, GLdouble );

    //OBJECTs
    Human *human;

    //VARIABLEs
    GLdouble
        base_1, base_2[2],
        secondJointAng[2];

    GLfloat
        viewRotX, viewRotY,
        h[16], d1, d2;

    SEGMENTS
        baseSegment, secondSegment, endSegment;
};

#endif

```

```

//*****
// FILE : InverseKinematics.cpp
// DESCRIPTION:
//*****

#include <math.h>
#include "InverseKinematics.h"

//-----
InverseKinematics::InverseKinematics(Human * man)
{
    // INITIALIZE
    human =man;
}

//-----
InverseKinematics::~InverseKinematics(){}
}

//-----
void InverseKinematics::setEarthOrientation( const GLfloat rotX,
                                              const GLfloat rotY)
{
    viewRotX =rotX;
    viewRotY =rotY;
}

//-----
void InverseKinematics::initialize( SEGMENTS end )
{
    if( end ==L_HAND ){

        baseSegment      =L_UPPER_ARM;
        secondSegment   =L_FORE_ARM;
        endSegment      =L_HAND;
    }
    else if( end ==R_HAND ){

        baseSegment      =R_UPPER_ARM;
        secondSegment   =R_FORE_ARM;
        endSegment      =R_HAND;
    }
    else if( end ==R FOOT ){

        baseSegment      =R_HIP;
        secondSegment   =R_LEG;
        endSegment      =R FOOT;
    }
    else if( end ==L FOOT ){

        baseSegment      =L_HIP;
        secondSegment   =L_LEG;
        endSegment      =L FOOT;
    }
    calculateSegmentLengths();
}

```

```

//-----
void InverseKinematics::algebraicSolution(const GLfloat x, const GLfloat y, const GLfloat z)
{
    boolean accepted =FALSE;
    calculateEndEffectorH_Matrix(x,y,z);

    if( endSegment ==L_HAND || endSegment ==R_HAND ){

        calculateAllAnglesForArm();
        GLint index=-1;

        do{ ++index;
            eulerToMatrix_YXZ( base_1, 0, base_2[index] );
            accepted =human ->setPosture( MATRIX, baseSegment, h);

            if( accepted ){
                h[Z] =(GLfloat) secondJointAng[index]/DEG_TO_RAD;
                accepted =human ->setPosture( EULER, secondSegment, h);
            }
        }while( !accepted && index <=0 );
    }
    else{
        calculateAllAnglesForLeg();
        GLint index=-1;

        do{ ++index;
            h[X] =(GLfloat) base_2[index]/DEG_TO_RAD;
            h[Y] =0;
            h[Z] =(GLfloat) base_1/DEG_TO_RAD;
            accepted =human ->setPosture( EULER, baseSegment, h);

            if( accepted ){
                h[X] =(GLfloat) secondJointAng[index]/DEG_TO_RAD;
                accepted =human ->setPosture( EULER, secondSegment, h);
            }
        }while( !accepted && index <=0 );
    }
}

//-----
void InverseKinematics::calculateSegmentLengths()
{
    d1 =human ->getSegmentLength( baseSegment );
    d2 =human ->getSegmentLength( secondSegment );
}

//-----
void InverseKinematics::calculateEndEffectorH_Matrix( const GLfloat x, const GLfloat y, const GLfloat z)
{
    GLfloat base_jc[THREE_D];
    human ->getJointCenters( baseSegment, base_jc[X], base_jc[Y], base_jc[Z] );

    h[0]=1;h[4]=0; h[8]=0;h[12]= base_jc[X] -x;
    h[1]=0;h[5]=1; h[9]=0;h[13]= base_jc[Y] -y;
    h[2]=0;h[6]=0;h[10]=1;h[14]= base_jc[Z] -z;
    h[3]=0;h[7]=0;h[11]=0;h[15]= 1;
}

```

```

glPushMatrix();
glLoadIdentity();
glRotatef( -viewRotX, 1, 0, 0 );
glRotatef( -viewRotY, 0, 1, 0 );
glMultMatrixf(h);
glGetFloatv( GL_MODELVIEW_MATRIX, h );
glPopMatrix();
}

//-----
void InverseKinematics::calculateAllAnglesForArm()
{
    GLdouble
        c1,s1,c2[2],s2[2],c3[2],s3[2];
    //Angle 1
    base_1 =atan2( h[14], h[13] );
    c1 =cos( base_1 );
    s1 =sin( base_1 );

    //Angle 2
    GLfloat
        d = (c1*h[13]) + (s1*h[14]),
        a = -2*d1*d,
        b = 2*h[12]*d1,
        c = (d2*d2) - (d1*d1) - (h[12]*h[12]) - (d*d),
        e =atan2(b,a),
        f =atan2( sqrt(fabs((a*a)+(b*b)-(c*c))) , c);

    base_2[0] = e + f;
    base_2[1] = e - f;

    for( GLint i=0; i<2; i++ ){

        c2[i] =cos( base_2[i] );
        s2[i] =sin( base_2[i] );
    }

    //Angle 3
    for( i=0; i<2; i++ ){

        s3[i] = ( (c2[i]*h[12]) + (s2[i]*c1*h[13]) + (s2[i]*s1*h[14]) ) / (-d2);
        c3[i] = ( (-s2[i]*h[12]) + (c2[i]*c1*h[13]) + (c2[i]*s1*h[14]) - d1 ) / d2;

        secondJointAng[i] = atan2( s3[i] , c3[i] );
    }
}

//-----
void InverseKinematics::calculateAllAnglesForLeg()
{
    GLdouble
        c1,s1,c2[2],s2[2],c3[2],s3[2];
    //Angle 1
    base_1 =-1*atan2( h[12], h[13] );
    c1 =cos( base_1 );

```

```

s1 =sin( base_1 );

//Angle 2
GLfloat
d = (s1*h[12]) - (c1*h[13]),
a = 2*d1*d,
b = -2*h[14]*d1,
c = (d2*d2) - (d1*d1) - (h[14]*h[14]) - (d*d),
e =atan2(b,a),
f =atan2( sqrt(fabs((a*a)+(b*b)-(c*c))) , c);

base_2[0] = e + f;
base_2[1] = e - f;

for( GLint i=0; i<2; i++ ){

    c2[i] =cos( base_2[i] );
    s2[i] =sin( base_2[i] );
}

//Angle 3
for( i=0; i<2; i++ ){

    s3[i] = ( (c2[i]*h[14]) - (s2[i]*c1*h[13]) + (s2[i]*s1*h[12]) ) / d2;
    c3[i] = ( (-s2[i]*h[14]) - (c2[i]*c1*h[13]) + (c2[i]*s1*h[12]) + d1 ) / (-d2);

    secondJointAng[i] = atan2( s3[i] , c3[i] );
}

}

//-----
void InverseKinematics::eulerToMatrix_YXZ( GLdouble angle_x, GLdouble angle_y, GLdouble angle_z)
{
    GLfloat
    cx =(GLfloat) cos( angle_x ),
    cy =(GLfloat) cos( angle_y ),
    cz =(GLfloat) cos( angle_z ),
    sx =(GLfloat) sin( angle_x ),
    sy =(GLfloat) sin( angle_y ),
    sz =(GLfloat) sin( angle_z ),

    cyz =cy*cz, syz =sy*sz, cszy =cz*sy, csyz =cy*sz;

    h[0] = cyz + (syz*sx);
    h[1] = cx*sz;
    h[2] = -cszy +(csyz*sx);

    h[3] = -csyz + (cszy*sx);
    h[4] = cx*cz;
    h[5] = syz + (cyz*sx);

    h[6] = sy*cx;
    h[7] = -sx;
    h[8] = cx*cy;
}

```

```

//*****
// FILE : Joint.h
// DESCRIPTION: Handling joint vertices
//*****
#ifndef __JOINT_H_
#define __JOINT_H_
#include <GL/glut.h>
#include <iostream.h>
#include "QuaternionR.h"
#include "utility.h"

const short NUMBER_OF_JOINTS =15; // number of humanJoint objects that are created

class Joint{
public :-----P U B L I C-----  

    //CONSTRUCTORS
    Joint( const GLint, GLfloat ** );
    Joint( Joint & );  

    //DESTRUCTOR
    ~Joint();  

    //FUNCTIONs
    void setRotation( const GLfloat ANGLE, const GLfloat VX,const GLfloat VY,const GLfloat VZ);
    void transformToEnd( GLfloat ** );// assign pointsAsEnd to associated points
    void transformToHead( GLfloat ** );// assign pointsAsHead to associated points
    boolean isConstructed();  

private :-----P R I V A T E-----  

    //CONSTs
    static const char JOINT_FILE_NAMES[NUMBER_OF_JOINTS][MAX_FILE_NAME];
    static const GLfloat END_TRANSLATIONS[NUMBER_OF_JOINTS][THREE_D];  

    //OPERATORS
    Joint& operator=( const Joint & );  

    //FUNCTIONs
    void transformPointsToHead();// transform joint points to end, set pointsAsEnd
    void transformPointsToEnd();// transform joint points to head, set pointsAsHead  

    //OBJECT POINTERS
    QuaternionR *rotation;  

    //VARIABLEs
    GLint
        numberOfJointPoints, //number of points that both segments have
        *indices; ///index numbers for these joint points
    GLfloat
        **jointPoints, **pointsAsEnd, **pointsAsHead, // initial and manipulated
        //joint points
        endTranslation[THREE_D], //translation values
        scaleFactor; //scaling factor
};

#endif

```

```

//*****
// FILE : Joint.cpp
// DESCRIPTION:
//*****

#include <math.h>
#include <string.h>
#include "Joint.h"

//*****INITIALIZE STATIC DATA MEMBERS *****
// file names of datas for index of points
const char Joint::JOINT_FILE_NAMES[NODE_OF_JOINTS][MAX_FILE_NAME] ={

    "Waist", "BodyNeck", "NeckHead",
    "L_hipJoint", "L_Knee", "L_Ankle",
    "L_Shoulder", "L_Elbow", "L_Wrist",
    "R_hipJoint", "R_Knee", "R_Ankle",
    "R_Shoulder", "R_Elbow", "R_Wrist"
};

// translation values of points
const GLfloat Joint::END_TRANSLATIONS[NODE_OF_JOINTS][THREE_D] ={

    { 0, 0.836942f, 0.145047f },//body
    { 0, 1.88292f, -0.145047f },//neck
    { 0, 0.391884f, 0.303044f },//head
    { 0.24161f, -0.349661f, -0.0217056f },//L_HIP
    { 0.1682516f, -1.60171f, -0.041123f },
    { -0.038406f, -1.88919f, -0.255575f },
    { 0.851806f, 1.64695f, -0.145047f },//L_UPPER_ARM
    { 0.358544f, -1.38687f, -0.2f },
    { 0.028006f, -1.10184f, 0.2f },
    { -0.24161f, -0.349661f, -0.0217056f },//R_HIP
    { -0.1682516f, -1.60171f, -0.041123f },
    { 0.038406f, -1.88919f, -0.255575f },
    { -0.851806f, 1.64695f, -0.145047f },//R_UPPER_ARM
    { -0.358544f, -1.38687f, -0.2f },
    { -0.028006f, -1.10184f, 0.2f }
};

//*****END STATIC DATA MEMBER INITIALIZATION *****

//-----
Joint::Joint( GLint jointNo , GLfloat ** points )
{
    scaleFactor      =1;
    numberOfJointPoints =0;
    jointPoints      =NULL;
    pointsAsHead     =NULL;
    pointsAsEnd      =NULL;
    rotation = new QuaternionR();

    // reading indices
    char fname[MAX_FILE_NAME];
    strcpy(fname, JOINT_FILE_NAMES[jointNo] );
    strcat(fname, ".dat" );
}

```

```

indices =readIndices(fname,numberOfJointPoints);

if( indices!= NULL ){
    // INITIALIZE joint points
    jointPoints =new GLfloat*[numberOfJointPoints];
    pointsAsHead =new GLfloat*[numberOfJointPoints];
    pointsAsEnd =new GLfloat*[numberOfJointPoints];

    for( GLint i=0; i<numberOfJointPoints ;i++ ){

        jointPoints[i] =new GLfloat[THREE_D];
        pointsAsHead[i] =new GLfloat[THREE_D];
        pointsAsEnd[i] =new GLfloat[THREE_D];

        for( GLint j=0; j<THREE_D; j++ ){

            jointPoints[i][j] =points[indices[i]][j];
        }
    }

    // SET translations
    for( i=0; i<THREE_D; i++ ){

        endTranslation[i] =END_TRANSLATIONS[jointNo][i];
    }
    transformPointsToHead();
    transformPointsToEnd();
}
}

//-----
Joint::~Joint()
{
    for( int i=0; i<numberOfJointPoints ;i++ ){

        delete [] jointPoints[i];
        delete [] pointsAsHead[i];
        delete [] pointsAsEnd[i];
    }
    delete [] jointPoints;
    delete [] pointsAsHead;
    delete [] pointsAsEnd;
    delete [] indices;
    delete rotation;
}

//-----
void Joint::transformToEnd( GLfloat ** points )
{
    for( GLint i=0; i<numberOfJointPoints ;i++ ){
        for( GLint j=0; j<THREE_D; j++ ){
            points[indices[i]][j]= pointsAsEnd[i][j];
        }
    }
}
}

```

```

//-----
void Joint::transformToHead( GLfloat ** points )
{
    for( GLint i=0; i<numberOfJointPoints ;i++ ){
        for( GLint j=0; j<THREE_D; j++ ){
            points[indices[i]][j]= pointsAsHead[i][j];
        }
    }
}

//-----
void Joint::setRotation( const GLfloat ANGLE, const GLfloat VX,const GLfloat VY,const GLfloat VZ)
{
    *rotation =QuaternionR( ANGLE, VX, VY, VZ );

    //scale factor depends on orientation
    scaleFactor =fabs ( cos( ANGLE * DEG_TO_RAD ) );

    transformPointsToHead();
    transformPointsToEnd();

    return;
}

//-----
void Joint::transformPointsToEnd()
{
    GLfloat tmp[THREE_D+1];

    //for all indices
    for( GLint i=0; i<numberOfJointPoints ;i++ ){

        //const. temp. point from initial joint points
        for( GLint j=0; j<THREE_D; j++ ){

            tmp[j] =jointPoints[i][j];
        }
        //rotate it forward
        QuaternionR tmpQ =rotation ->rotate(QuaternionR(tmp[X],tmp[Y],tmp[Z]));

        tmpQ.getValues( tmp );
        pointsAsEnd[i][X]= (tmp[X]*scaleFactor) + endTranslation[X];
        pointsAsEnd[i][Y]= (tmp[Y]*scaleFactor) + endTranslation[Y];
        pointsAsEnd[i][Z]= (tmp[Z]*scaleFactor) + endTranslation[Z];
    }
    return;
}

//-----
void Joint::transformPointsToHead()
{
    GLfloat tmp[THREE_D+1];

    //for all indices
    for( GLint i=0; i<numberOfJointPoints ;i++ ){

```

```

//const. temp. point from initial joint points
for( GLint j=0; j<THREE_D; j++ ){

    tmp[j] =jointPoints[i][j];
}
// get inverse of quaternion
QuaternionR inverseRotation =- (*rotation);

//rotate point backward
QuaternionR tmpQ =inverseRotation.rotate(QuaternionR(tmp[X],tmp[Y],tmp[Z]));

tmpQ.getValues( tmp );
pointsAsHead[i][X]= (tmp[X]*scaleFactor);
pointsAsHead[i][Y]= (tmp[Y]*scaleFactor);
pointsAsHead[i][Z]= (tmp[Z]*scaleFactor);
}
return;
}

//-----
boolean Joint::isConstructed()
{
    boolean status =FALSE;

    if ( indices !=NULL ){

        status =TRUE;
    }

    return status;
}

```

```

//*****
// FILE : KeyframeAnim.h
// DESCRIPTION: Functions to handle frame list
//*****
#ifndef __KEYFRAMEANIM_H__
#define __KEYFRAMEANIM_H__
#include <GL/glut.h>
#include "utility.h"
#include "Posture.h"

class KeyFrameAnim{

public :-----P U B L I C-----
    //CONSTRUCTORS
    KeyFrameAnim( Human * );                                //default
    KeyFrameAnim( KeyFrameAnim & ); //copy

    //DESTRUCTOR
    ~KeyFrameAnim();

    //FUNCTIONs
    void keyPressed( GLint key );
    boolean play();

private :-----P R I V A T E-----
    //OPERATORS
    KeyFrameAnim& operator=( const KeyFrameAnim & );

    //FUNCTIONs
    void add();
    void insert();
    void remove();
    void next();
    void previous();
    void start();
    void stop();
    void gotoFirst();
    void gotoLast();

    void setPosture( Posture * );
    void getPosture( Posture * );

    boolean switchToNextKey();
    void reduceInterpolation();
    void interpolate();

    //OBJECTs
    Posture *first, *last, *current,*iterator, *nextIterator;
    Human *human;

    //VARIABLEs
    GLint numberOfUnmatch;
    GLfloat time, orientation[THREE_D+1];
    SEGMENTS unmatch[ NUMBER_OF_SEGMENTS ];

};

#endif

```

```

//*****
// FILE    : KeyFrameAnim.cpp
// DESCRIPTION:
//*****

#include "KeyFrameAnim.h"

//-----
KeyFrameAnim::KeyFrameAnim( Human * man ){

    // INITIALIZE
    human =man;
    iterator =nextIterator =first =last =current =NULL;

    time =0;
}

//-----
KeyFrameAnim::~KeyFrameAnim(){

}

//-----
void KeyFrameAnim::keyPressed( GLint key )
{
    switch( key ){

        case 'a' :add();
                    break;
        case 'n' :next();
                    break;
        case 'p' :previous();
                    break;
        case 'd' :remove();
                    break;
        case 'i' :insert();
                    break;
        case 'r' :start();
                    break;
        case 's' :stop();
                    break;
        case 'f' :gotoFirst();
                    break;
        case 'l' :gotoLast();
                    break;
    };
}

//-----
void KeyFrameAnim::setPosture( Posture * p )
{
    for(GLint segmentNo=0; segmentNo< NUMBER_OF_SEGMENTS; segmentNo++){

        human ->setPosture( QUATERNION, (SEGMENTS)segmentNo,
        p ->getQuaternion((SEGMENTS)segmentNo) );
    }
}

```

```

//-----
void KeyFrameAnim::getPosture( Posture * const p )
{
    for(GLint segmentNo=0; segmentNo< NUMBER_OF_SEGMENTS; segmentNo++){

        human ->getPosture( QUATERNION, (SEGMENTS)segmentNo,
        p ->getQuaternion((SEGMENTS)segmentNo) );
    }
}

//-----
void KeyFrameAnim::add()
{
    if( first ==NULL ){

        current =last =first =new Posture();
        getPosture( first );
    }
    else{

        Posture *tmp =new Posture();
        getPosture( tmp );
        last ->addNextPosture( tmp );
        current =last =tmp;
    }
}

//-----
void KeyFrameAnim::insert()
{
    if( first ==NULL ){

        add();
    }
    else{

        Posture *tmp =new Posture();
        getPosture( tmp );
        tmp ->addNextPosture( current ->getNextPosture() );
        current ->addNextPosture( tmp );
        if( last ==current ){

            last =tmp;
        }
        current =tmp;
    }
}

//-----
void KeyFrameAnim::remove()
{
    if( first ==NULL ){ //do nothing
    }
}

```

```

else if( current ==first ){

    if( last == first ){

        last =first =NULL;
        human ->initializePosture();
    }
    else{
        first =first ->getNextPosture();
        setPosture( first );
    }

    delete current;
    current =first;
}
else{
    Posture* tmp=first;

    while( tmp ->getNextPosture() != current ){
        tmp =tmp ->getNextPosture();
    }

    tmp ->addNextPosture( current->getNextPosture() );
    if( last ==current ){

        last =tmp;
    }
    delete current;
    current =tmp;
    setPosture( current );
}
}

//-----
void KeyFrameAnim::next()
{
    if( current != last ){

        current =current ->getNextPosture();
        setPosture( current );
    }
}

//-----
void KeyFrameAnim::previous()
{
    if( current != first ){

        Posture* tmp=first;

        while( tmp ->getNextPosture() != current ){
            tmp =tmp ->getNextPosture();
        }
        current =tmp;
        setPosture( current );
    }
}

```

```

//-----
void KeyFrameAnim::start()
{
    time =0;
    numberOfUnmatch =0;
    iterator =first;
    nextIterator =first ->getNextPosture();
    setPosture( first );
    reduceInterpolation();
}

//-----
boolean KeyFrameAnim::play()
{
    boolean running =TRUE;
    if( time <1 ){
        interpolate();
    }
    else{
        time =0;
        running =switchToNextKey();
    }
    return running;
}

//-----
boolean KeyFrameAnim::switchToNextKey()
{
    boolean running =FALSE;
    if( nextIterator!= last ){
        iterator =iterator ->getNextPosture();
        nextIterator =iterator ->getNextPosture();
        reduceInterpolation();
        interpolate();
        running =TRUE;
    }
    return running;
}

```

```

//-----
void KeyFrameAnim::reduceInterpolation()
{
    for(GLint segmentNo=0; segmentNo< NUMBER_OF_SEGMENTS; segmentNo++){

        GLfloat
            *q1 =iterator ->getQuaternion((SEGMENTS)segmentNo),
            *q2 =nextIterator ->getQuaternion((SEGMENTS)segmentNo);

        if( q1[X]!=q2[X] || q1[Y]!=q2[Y] || q1[Z]!=q2[Z] || q1[3]!=q2[3] ){

            unmatch[ numberOfUnmatch++ ] =(SEGMENTS)segmentNo;
        }
    }
}

//-----
void KeyFrameAnim::interpolate()
{
    static const GLfloat DELTA_T =0.1f;

    for(GLint i=0; i<numberOfUnmatch; i++){

        quatInterpolation( iterator ->getQuaternion(unmatch[i]),
                            nextIterator ->getQuaternion(unmatch[i]),
                            time,
                            orientation );

        human ->setPosture( QUATERNION, unmatch[i], orientation );
    }
    time +=DELTA_T;
}

//-----
void KeyFrameAnim::stop()
{
}

//-----
void KeyFrameAnim::gotoFirst()
{
    current =first;
    setPosture( current );
}

//-----
void KeyFrameAnim::gotoLast()
{
    current =last;
    setPosture( current );
}

```

```

***** : Posture.h
// DESCRIPTION: Nodes of Frame List
*****


#ifndef __POSTURE_H__
#define __POSTURE_H__


#include <GL/glut.h>
#include "utility.h"
#include "Human.h"

class Posture{

public :-----P U B L I C-----


//CONSTRUCTORS
Posture(); //default
Posture( Posture & ); //copy

//DESTRUCTOR
~Posture();

//FUNCTIONs
void setQuaternion( const SEGMENTS, const GLfloat * const );
GLfloat* getQuaternion( const SEGMENTS );

void addNextPosture( Posture * const );
Posture * getNextPosture();

private :-----P R I V A T E-----


//OPERATORS
Posture& operator=( const Posture & );

//FUNCTIONs

//OBJECTs
Posture *next;

//VARIABLEs
GLfloat segmentQuaternion[NUMBER_OF_SEGMENTS][THREE_D+1];

};

#endif

```

```

//*****
// FILE      : Posture.cpp
// DESCRIPTION:
//*****


#include "Posture.h"

//-----
Posture::Posture(){

    // INITIALIZE
    next=NULL;
}

//-----
Posture::~Posture(){
}

//-----
void Posture::setQuaternion( const SEGMENTS NO, const GLfloat * const QUAT)
{
    for( GLint i=0; i<(THREE_D+1); i++ ){

        segmentQuaternion[ NO ][i] =QUAT[i];
    }
}

//-----
GLfloat* Posture::getQuaternion( const SEGMENTS NO )
{
    return segmentQuaternion[ NO ];
}

//-----
void Posture::addNextPosture( Posture * const pos )
{
    next =pos;
}

//-----
Posture * Posture::getNextPosture()
{
    return next;
}

```

```

***** : ProceduralAnim.h
// DESCRIPTION: Forward & Inverse Kinematics Walking procedures
*****  

#ifndef __PROCEDURALANIM_H__
#define __PROCEDURALANIM_H__
#include <GL/glut.h>
#include "Human.h"
#include "utility.h"
#include "QuaternionR.h"
enum KINEMATIC_TYPE { INVERSE, FORWARD };  

class ProceduralAnim{
public :-----P U B L I C-----  

    //CONSTRUCTORS
    ProceduralAnim( Human * );           //default
    ProceduralAnim( ProceduralAnim & ); //copy  

    //DESTRUCTOR
    ~ProceduralAnim();  

    //CONSTs
    static const GLfloat MAX_FRAME_RATE, MIN_FRAME_RATE, FRAME_ACC;  

    //FUNCTIONs
    void increaseFrameRate();
    void decreaseFrameRate();
    void setWalkingMethod( KINEMATIC_TYPE );
    void walk();  

private :-----P R I V A T E-----  

    //CONSTs
    static const GLfloat SUPPORT_PHASE,
                      TIME_2, TIME_2B, TIME_3, TIME_3B, TIME_4, TIME_5;
    //OPERATORS
    ProceduralAnim& operator=( const ProceduralAnim & );  

    //FUNCTIONs
    void walk_ik(); //runs walking cycle
    void walk_fk();
    void step_ik( const SEGMENTS , const SEGMENTS , const GLfloat TIME);
    void step_fk( const SEGMENTS , const SEGMENTS ,const SEGMENTS , const GLfloat TIME);
    void synchronizeArm( const SEGMENTS hip,
                         const GLfloat HIP_ANGLE, const GLfloat KNEE_ANGLE );
    void walkingBodyMotion();
    GLfloat hipMotion( const GLfloat );
    GLfloat kneeMotion( const GLfloat );
    GLfloat ankleMotion( const GLfloat );  

    //OBJECT POINTERS
    Human * human;  

    //VARIABLEs
    GLfloat orientation[ THREE_D+1 ], time, frameRate;
    KINEMATIC_TYPE walkingMethod;  

};  

#endif

```

```

//*****
// FILE    : ProceduralAnim.cpp
// DESCRIPTION:
//*****

#include "ProceduralAnim.h"

//*****INITIALIZE STATIC DATA MEMBERS *****

const GLfloat
    ProceduralAnim::MAX_FRAME_RATE    =0.9f,
    ProceduralAnim::MIN_FRAME_RATE    =0.1f,
    ProceduralAnim::FRAME_ACC        =0.1f,
    ProceduralAnim::SUPPORT_PHASE    =(GLfloat)PI,
    ProceduralAnim::TIME_2            =(GLfloat)PI / 4,
    ProceduralAnim::TIME_3            =(GLfloat)PI,
    ProceduralAnim::TIME_2B           =TIME_3 - (SUPPORT_PHASE/2),
    ProceduralAnim::TIME_3B           =TIME_3 + (SUPPORT_PHASE/2),
    ProceduralAnim::TIME_4            =7*(GLfloat)PI / 4,
    ProceduralAnim::TIME_5            =2*(GLfloat)PI;

//*****END STATIC DATA MEMBER INITIALIZATION *****

//-----
ProceduralAnim::ProceduralAnim( Human * man ){

    // INITIALIZE
    human =man;
    time =0;
    walkingMethod =FORWARD;
    frameRate =FRAME_ACC;
}

//-----
ProceduralAnim::~ProceduralAnim(){
}

//-----
void ProceduralAnim::setWalkingMethod( KINEMATIC_TYPE method )
{
    walkingMethod =method;
}

//-----
void ProceduralAnim::increaseFrameRate()
{
    frameRate +=FRAME_ACC;

    if( frameRate >MAX_FRAME_RATE ) frameRate =MAX_FRAME_RATE;

    return;
}

```

```

//-----
void ProceduralAnim::decreaseFrameRate()
{
    frameRate -= FRAME_ACC;

    if( frameRate < MIN_FRAME_RATE ) frameRate = MIN_FRAME_RATE;

    return;
}

//-----
void ProceduralAnim:: walk()
{
    if( human != NULL ){

        ( walkingMethod == FORWARD ) ? walk_fk() : walk_ik();
    }
}

//-----
void ProceduralAnim::walk_ik()
{
    ( time > TIME_5 ) ? time = frameRate : time+=frameRate;

    step_ik( R_HIP, R_LEG, time );

    GLfloat ttime = time+TIME_3;
    if( ttime > TIME_5 ) ttime-=TIME_5;
    step_ik( L_HIP, L_LEG, ttime );

    walkingBodyMotion();

    return;
}

//-----
void ProceduralAnim::step_ik( const SEGMENTS upLeg,
                             const SEGMENTS lowLeg,
                             const GLfloat TIME )
{
    static const GLfloat MAX_STEP_ANGLE = -20;

    GLfloat upLegLength = human->getSegmentLength(R_HIP),
           lowLegLength = human->getSegmentLength(R_LEG),
           legLength = upLegLength + lowLegLength;

    // calc. pathAngle
    GLfloat pathAngle = MAX_STEP_ANGLE * sin( TIME );

    // calc. pathRadius
    GLfloat pathRadius = legLength * cos( pathAngle * DEG_TO_RAD );

    // Recovery Leg Motion ( FORWARD step )
    if( TIME < TIME_2 || TIME > TIME_4 ){
        pathRadius *= 0.95f;
    }
}

```

```

// calc. (y,z) for end-effector
GLfloat y = pathRadius * cos( pathAngle * DEG_TO_RAD ),
z = pathRadius * sin( pathAngle * DEG_TO_RAD ),
hipAngle, kneeAngle;

// calc. angles
twoLink2D( z, y,
            upLegLength, lowLegLength,
            hipAngle, kneeAngle );
// motion
orientation[X] =1;
orientation[Y] =0;
orientation[Z] =0;
orientation[3] =hipAngle;
human -> setPosture( VECTOR_ANGLE, upLeg, orientation );
orientation[3] =kneeAngle;
human -> setPosture( VECTOR_ANGLE, lowLeg, orientation );

synchronizeArm( upLeg, hipAngle, kneeAngle );
}

//-----
void ProceduralAnim::walk_fk()
{
    ( time > TIME_5 ) ? time =frameRate : time+=frameRate;

    step_fk( R_HIP, R_LEG, R FOOT , time );

    GLfloat ttime =time+TIME_3;
    if( ttime > TIME_5 ) ttime-=TIME_5;
    step_fk( L_HIP, L_LEG, L FOOT , ttime );

    walkingBodyMotion();
}

//-----
void ProceduralAnim::step_fk( const SEGMENTS upLeg, const SEGMENTS lowLeg,
                           const SEGMENTS foot, const GLfloat TIME)
{
    GLfloat hipAngle =hipMotion(TIME);
    orientation[X] =-1;
    orientation[Y] =0;
    orientation[Z] =0;
    orientation[3] =hipAngle;
    human -> setPosture( VECTOR_ANGLE, upLeg, orientation );

    GLfloat kneeAngle =kneeMotion(TIME);
    orientation[3] =kneeAngle;
    human -> setPosture( VECTOR_ANGLE, lowLeg, orientation );

    GLfloat footAngle =ankleMotion(TIME);
    orientation[3] =footAngle;
    human -> setPosture( VECTOR_ANGLE, foot, orientation );
    synchronizeArm( upLeg, -hipAngle, -kneeAngle );
}

```

```

//-----
GLfloat ProceduralAnim::hipMotion( const GLfloat TIME )
{
    static const GLfloat HALF_PI    =(GLfloat) PI/2;

    GLfloat result;

    if( TIME <=TIME_2 ){

        GLfloat t =linearInterpolate( HALF_PI, TIME_2, TIME );
        result = 45 * sin( t );
    }
    else if( TIME >TIME_2 && TIME <=TIME_3 ){

        GLfloat t =linearInterpolate( HALF_PI, TIME_3 -TIME_2 , TIME -TIME_2 );
        result = 45 * sin( t +HALF_PI );
    }
    else if( TIME >TIME_3 && TIME <=TIME_4 ){

        GLfloat t =linearInterpolate( HALF_PI, TIME_4 -TIME_3 , TIME -TIME_3 );
        result = -35 * sin( t );
    }
    else{

        GLfloat t =linearInterpolate( HALF_PI, TIME_5 - TIME_4, TIME - TIME_4 );
        result = -35 * sin( t +HALF_PI );
    }

    return result;
}

//-----
GLfloat ProceduralAnim::kneeMotion( const GLfloat TIME )
{
    static const GLfloat
        MAX_ANGLE  =-35;

    GLfloat result;

    if( TIME <=TIME_2B ){

        GLfloat t =linearInterpolate( (GLfloat)PI, TIME_2B, TIME );
        result =MAX_ANGLE * sin( t );
    }
    else if( TIME >=TIME_3B ){

        GLfloat t =linearInterpolate( (GLfloat)PI, TIME_5 - TIME_3B, TIME - TIME_3B );
        result =MAX_ANGLE * sin( t );
    }
    else{

        result =0;
    }

    return result;
}

```

```

//-----
GLfloat ProceduralAnim::ankleMotion( const GLfloat TIME )
{
    GLfloat result;

    if( TIME < TIME_2B ){

        GLfloat t =linearInterpolate( (GLfloat)PI, TIME_2B, TIME );
        result =-20 * sin( t );
    }
    else if( TIME >= TIME_2B && TIME <= TIME_3 ){

        result =0;
    }
    else if( TIME > TIME_3 && TIME < TIME_3B ){

        GLfloat t =linearInterpolate( (GLfloat)PI, TIME_3B - TIME_3 , TIME - TIME_3 );
        result =10 * sin( t );
    }
    else{

        GLfloat t =linearInterpolate( (GLfloat)PI, TIME_5 - TIME_3B , TIME - TIME_3B );
        result =-5 * sin( t );
    }

    return result;
}

//-----
void ProceduralAnim::synchronizeArm( const SEGMENTS upLeg,
                                      const GLfloat HIP_ANGLE, const GLfloat KNEE_ANGLE )
{
    static const GLfloat
        HIP_TO_SHOULDER =0.7f,
        KNEE_TO_ELBOW =0.3f;
    // move arm (no computation, synchronize with leg, by using some constant vals.)
    SEGMENTS upArm =R_UPPER_ARM,
             lowArm =R_FORE_ARM;

    if( upLeg ==R_HIP ){

        upArm =L_UPPER_ARM;
        lowArm =L_FORE_ARM;
    }

    //up arm synchronization
    orientation[X] =1;
    orientation[Y] =0;
    orientation[Z] =0;
    orientation[3] =HIP_ANGLE * HIP_TO_SHOULDER;
    human -> setPosture( VECTOR_ANGLE, upArm, orientation );
}

```

```

if( KNEE_ANGLE<0 ){
    //lower arm synchronization if angle is neg.
    orientation[3] =-KNEE_ANGLE * KNEE_TO_ELBOW;
    human -> setPosture( VECTOR_ANGLE, lowArm, orientation );
}
else{
    //lower arm synchronization if angle is pos.
    orientation[3] =0;
    human -> setPosture( VECTOR_ANGLE, lowArm, orientation );
}
return;
}

//-----
void ProceduralAnim::walkingBodyMotion()
{
    GLfloat pelvicRot =5 *sin( time );
    orientation[X] =0;
    orientation[Y] =1;
    orientation[Z] =0;
    orientation[3] =pelvicRot;
    human -> setPosture( VECTOR_ANGLE, ROOT, orientation );

    GLfloat pelvicTilt =4 *sin( time );
    orientation[Y] =0;
    orientation[Z] =1;
    orientation[3] =pelvicTilt;
    human -> setPosture( VECTOR_ANGLE, BODY, orientation );

    // whole body translation ( h = -0.08 * sin(2 pi f t) , here f=1/2*PI
    glTranslatef(0, ( -0.08f * sin( fabs(time-PI) ) ), 0 );
}

return;
}

```

```

***** : Quaternion.h
// DESCRIPTION: functions for quaternion algebra
*****  

#ifndef __QUATERNIONR_H__
#define __QUATERNIONR_H__
#include <GL/glut.h>
#include <iostream.h>
#include <iomanip.h>
#include "utility.h"
#define PI 3.14159265358979323846

class QuaternionR{

    // overloaded operator<<
    friend ostream &operator<<(ostream &,const QuaternionR &);

public :-----P U B L I C-----
    //CONSTRUCTORS
    QuaternionR ();           //default
    QuaternionR (const QuaternionR &); //copy
    QuaternionR (const GLfloat VX,const GLfloat VY,const GLfloat VZ, const GLfloat ANGLE);
    QuaternionR (const GLfloat * );
    QuaternionR (const GLfloat QX,const GLfloat QY,const GLfloat QZ);

    //DESTRUCTOR
    ~QuaternionR();

    //OPERATORS
    // quaternion product
    QuaternionR operator*(const QuaternionR &) const;
    // Quaternion addition
    QuaternionR &operator*=(const QuaternionR &);
    // quaternion inverse (conjugate)
    QuaternionR operator+(const QuaternionR &) const;
    // quaternion assignment
    QuaternionR &operator=(const QuaternionR &);
    // quaternion product and assignment
    QuaternionR operator-();

    //FUNCTIONS
    void QuaternionR::setValues(const GLfloat *);
    void QuaternionR::getValues(GLfloat * const);
    // rotate a quaternion about a 3D vector (w=0)
    QuaternionR rotate(const QuaternionR &);
    // dot product
    QuaternionR dotProduct(const QuaternionR &);
    // quaternion to axis angles
    void normalize();

private :-----P R I V A T E-----
    //VARIABLEs
    GLfloat w,x,y,z;
};

#endif

```

```

//*****
// FILE      : QuaternionR.cpp
// DESCRIPTION:
//*****


#include <math.h>
#include "QuaternionR.h"


//-----
QuaternionR::QuaternionR()
{
    w= 1;
    x= y= z= 0;
}

//-----
QuaternionR::QuaternionR(const GLfloat *vecAng )
{
    *this =QuaternionR( vecAng[X], vecAng[Y],vecAng[Z],vecAng[3] );
}

//-----
QuaternionR::QuaternionR(const GLfloat VX,const GLfloat VY,const GLfloat VZ,
                           const GLfloat ANGLE )
{
    GLfloat sinAngle =( GLfloat ) sin( ANGLE*DEG_TO_RAD/2 );

    x =VX * sinAngle;
    y =VY * sinAngle;
    z =VZ * sinAngle;

    w =( GLfloat ) cos( ANGLE*DEG_TO_RAD/2 );

    normalize();
}

//-----
QuaternionR::QuaternionR(const GLfloat QX,const GLfloat QY,const GLfloat QZ)
{
    x =QX;
    y =QY;
    z =QZ;
    w =0;
}

//-----
QuaternionR::~QuaternionR()
{
}

//-----
QuaternionR::QuaternionR(const QuaternionR &QUAT)
{
    w =QUAT.w;
    x =QUAT.x;
}

```

```

y = QUAT.y;
z = QUAT.z;
}

//-----
QuaternionR & QuaternionR::operator=(const QuaternionR &QUAT)
{
    w = QUAT.w;
    x = QUAT.x;
    y = QUAT.y;
    z = QUAT.z;

    return (*this);
}

//-----
QuaternionR QuaternionR::operator*(const QuaternionR &QUAT) const
{
    QuaternionR dest;

    dest.w =      QUAT.w * w - QUAT.x * x - QUAT.y * y - QUAT.z * z;
    dest.x =      QUAT.w * x + QUAT.x * w - QUAT.y * z + QUAT.z * y;
    dest.y =      QUAT.w * y + QUAT.y * w - QUAT.z * x + QUAT.x * z;
    dest.z =      QUAT.w * z + QUAT.z * w - QUAT.x * y + QUAT.y * x;

    return(dest);
}

//-----
QuaternionR & QuaternionR::operator*=(const QuaternionR &QUAT)
{
    *this = *this * QUAT;

    return (*this);
}

//-----
QuaternionR QuaternionR::operator+(const QuaternionR &QUAT) const
{
    QuaternionR add;

    add.w = w + QUAT.w;
    add.x = x + QUAT.x;
    add.y = y + QUAT.y;
    add.z = z + QUAT.z;

    return (add);
}

//-----
QuaternionR QuaternionR::operator-()
{
    QuaternionR temp;

    temp.w = w;
    temp.x = -x;
}

```

```

temp.y = -y;
temp.z = -z;

return (temp);
}

//-----
QuaternionR QuaternionR::rotate(const QuaternionR &QUAT)
{
    QuaternionR temp;

    temp = *this * ( QUAT * (-(*this)));

    return (temp);
}

//-----
QuaternionR QuaternionR::dotProduct(const QuaternionR &QUAT)
{
    QuaternionR temp;

    temp.w = w + QUAT.w;
    temp.x = x + QUAT.x;
    temp.y = y + QUAT.y;
    temp.z = z + QUAT.z;

    return (temp);
}
//-----
void QuaternionR::normalize()
{
    GLfloat magnitude;

    magnitude = sqrt( (x * x) + (y * y) + (z * z) + (w * w) );

    if( magnitude > 1 ){

        x = x / magnitude;
        y = y / magnitude;
        z = z / magnitude;
        w = w / magnitude;
    }
    return;
}

//-----
ostream &operator<<(ostream &output, const QuaternionR &q)
{
    output << '['
        << " w_" << q.w
        << " x_" << q.x
        << " y_" << q.y
        << " z_" << q.z
        << ']';
    return output;
}

```

```
-----  
void QuaternionR::setValues(const GLfloat *val )  
{  
    x =val[X];  
    y =val[Y];  
    z =val[Z];  
    w =val[3];  
}  
  
-----  
void QuaternionR::getValues( GLfloat * const val )  
{  
    val[X] =x;  
    val[Y] =y;  
    val[Z] =z;  
    val[3] =w;  
}
```

```

//***** : Segment.h
// DESCRIPTION: Individual segment functions
//***** 

#ifndef __SEGMENT_H__
#define __SEGMENT_H__

#include <GL/glut.h>
#include <iostream.h>
#include "utility.h"
#include "TriangleFaceSet.h"
#include "H_Matrix3DOF.h"
#include "H_Matrix1DOF.h"
#include "Joint.h"

//ENUMs
enum MODEL_TYPE{ STICK, SKIN, WIRE_FRAME }; //model types

//CONSTs
const GLshort NUMBER_OF_SEGMENTS =16; // number of humanSegment objects that are created

//CLASS DEF
class Segment{

public ://-----P U B L I C-----


//CONSTRUCTORS
Segment( const GLint SEGMENT_NO, const AXIS DOF );
Segment( Segment & );

//DESTRUCTOR
~Segment();

//FUNCTIONS
boolean constructShape( GLfloat **, Joint * );

void draw( GLfloat **, GLfloat ** );
void drawDetectionVolume();

void setModelType( MODEL_TYPE );
void setStickShape( Box& );

boolean setOrientation( const ROTATION_METHODS ,const GLfloat* );
void getOrientation( const ROTATION_METHODS , GLfloat * const );
boolean rotate( const ROTATION_METHODS, const GLfloat * );

void getJointCenters(GLfloat &, GLfloat &, GLfloat &);

void getEarthMatrix( GLfloat[16] );
GLfloat getLength();

```

```

private :-----P R I V A T E-----  

//CONSTs
    // file names of datas for index of points
    static const char
        SEGMENT_FILE_NAMES[NODE_OF_SEGMENTS][MAX_FILE_NAME];
    // translation values of points
    static const GLfloat TRANSLATIONS[NODE_OF_SEGMENTS][THREE_D];
    // constraint of segment orientation
    static const GLfloat ROTATION_BOUNDARIES[NODE_OF_SEGMENTS][2*THREE_D];  

//OPERATORS
Segment& operator=( const Segment & );  

//FUNCTIONs
void drawJointBall();  

//OBJECT POINTERS
TriangleFaceSet * shape;
Box      * stick;
H_Matrix * localMatrix;
Joint    * jointPoints;  

//VARIABLEs
MODEL_TYPE
    modelType;// state var. for drawing
GLint
    segmentNo; //
GLfloat
    modelViewMatrix[16]; //model view matrix of system when this segment is drawn
};  

#endif

```

```

//*****
// FILE : Segment.cpp
// DESCRIPTION:
//*****

#include <math.h>
#include <string.h>
#include "Segment.h"

//*****INITIALIZE STATIC DATA MEMBERS *****
// file names of datas for index of points
const char
Segment::SEGMENT_FILE_NAMES[NODE_OF_SEGMENTS][MAX_FILE_NAME] ={

    "Root", "Body", "Neck", "Head",
    "L_Hip", "L_Leg", "L_Foot",
    "L_UpperArm", "L_ForeArm", "L_Hand",
    "R_Hip", "R_Leg", "R_Foot",
    "R_UpperArm", "R_ForeArm", "R_Hand"
};

// position values of points
const GLfloat Segment::TRANSLATIONS[NODE_OF_SEGMENTS][THREE_D] ={

    { 0, 0, 0 },//root
    { 0, 0.836942f, 0.145047f },//body
    { 0, 1.88292f, -0.145047f },//neck
    { 0, 0.391884f, 0.303044f },//head
    { 0.24161f, -0.349661f, -0.0217056f },//L_HIP
    { 0.1682516f, -1.60171f, -0.041123f },
    { -0.038406f, -1.88919f, -0.255575f },
    { 0.851806f, 1.64695f, -0.145047f },//L_UPPER_ARM
    { 0.358544f, -1.38687f, -0.2f },
    { 0.028006f, -1.10184f, 0.2f },
    { -0.24161f, -0.349661f, -0.0217056f },//R_HIP
    { -0.1682516f, -1.60171f, -0.041123f },
    { 0.038406f, -1.88919f, -0.255575f },
    { -0.851806f, 1.64695f, -0.145047f },//R_UPPER_ARM
    { -0.358544f, -1.38687f, -0.2f },
    { -0.028006f, -1.10184f, 0.2f }
};

// constraint of segment orientation
const GLfloat
Segment::ROTATION_BOUNDARIES[NODE_OF_SEGMENTS][2*THREE_D] ={

    //Y =elevation, z =azimuth, x =nose
    //{{MinY,MaxY,MinZ,MaxZ,MinX,MaxX }
    {0,0,0,0,0,0}, //root
    {-40,160,-95,95,-30,30}, //body
    {-30,45,-50,50,-30,30}, //neck
    {-30,50,-45,45,-30,30}, //head
    {-100,50,-30,30,-10,90}, //L_HIP
    {0,120,0,0,0,0},
    {-30,70,-20,20,-20,20},
    {-180,80,-90,90,-30,180}, //L_UPPER_ARM
};

```

```

{0,0,0,-140,0},
{-30,30,-60,60,-80,80},
{-100,50,-30,30,-90,10}, //R_HIP
{0,120,0,0,0,0},
{-30,70,-20,20,-20,20},
{-180,80,-90,90,-180,30}, //R_UPPER_ARM
{0,0,0,0,140},
{-30,30,-60,60,-80,80}
};

//*****END STATIC DATA MEMBER INITIALIZATION *****

//-----
Segment::Segment( const GLint SEGMENT_NO, const AXIS DOF )
{
    segmentNo =SEGMENT_NO;

    modelType =STICK;

    shape      =NULL;
    jointPoints =NULL;

    //CREATE H_Matrix
    if( DOF <=Z ){
        localMatrix =new H_Matrix1DOF(DOF);
    }
    else if( DOF ==XYZ ){
        localMatrix =new H_Matrix3DOF();
    }
    localMatrix ->setBoundaries( ROTATION_BOUNDARIES[ SEGMENT_NO ] );

    // SET positions
    localMatrix->setTranslation( TRANSLATIONS[segmentNo][X],
                                  TRANSLATIONS[segmentNo][Y],
                                  TRANSLATIONS[segmentNo][Z] );

    // SET
    if( segmentNo+1 < NUMBER_OF_SEGMENTS ){

        stick =new Box( TRANSLATIONS[segmentNo+1][X],
                        TRANSLATIONS[segmentNo+1][Y],
                        TRANSLATIONS[segmentNo+1][Z]);
    }
    else{
        stick =new Box(0,0,0);
    }
}

//-----
Segment::~Segment()
{
    delete shape;
    delete localMatrix;
    delete jointPoints;
    delete stick;
}

```

```

//-----
boolean Segment::constructShape( GLfloat ** points, Joint * joint )
{
    // reading indices
    char fname[MAX_FILE_NAME];

    strcpy(fname, SEGMENT_FILE_NAMES[segmentNo] );
    strcat(fname, ".dat" );

    shape =new TriangleFaceSet(fname,points);

    boolean status =shape ->isConstructed();

    if( status ==TRUE ){

        jointPoints =joint;
    }

    return status;
}

//-----
void Segment::draw( GLfloat ** points, GLfloat ** normals )
{
    // make translation and orientation to set posture
    localMatrix ->applyToCurrentMatrix();

    //draw model
    switch( modelType ){

        case STICK : glEnable(GL_LIGHTING);
                      drawJointBall();
                      glColor3f(0.5f,0.5f,1);
                      stick ->draw();
                      break;

        case SKIN : glEnable(GL_LIGHTING);
                     glPolygonMode(GL_FRONT, GL_FILL);
                     shape ->drawTriangles( points, normals );
                     break;

        case WIRE_FRAME :glDisable(GL_LIGHTING);
                           drawJointBall();
                           glPolygonMode(GL_FRONT, GL_LINE);
                           glColor3f(1,1,1);
                           shape ->drawTriangles( points, normals );
                           break;

        default :
            break;
    }

    //hold MODEL VIEW matrix
    glGetFloatv( GL_MODELVIEW_MATRIX, modelViewMatrix );

    return;
}

```

```

//-----
void Segment::drawJointBall()
{
    glColor3f(1,0,0);
    glPolygonMode(GL_FRONT, GL_FILL);
    glutSolidSphere(0.1f,10,10);
}

//-----
void Segment::drawDetectionVolume()
{
    //set view matrix of this segment
    glLoadMatrixf( modelViewMatrix );
    ( modelType == STICK ) ? stick ->show() : shape ->showBounds();
}

//-----
boolean Segment::setOrientation( const ROTATION_METHODS method,
                                const GLfloat *orientation )
{
    boolean rotationAccepted = localMatrix ->setOrientation( method, orientation );

    if( jointPoints != NULL ){

        GLfloat vecAng[THREE_D+1];
        localMatrix ->getOrientation( VECTOR_ANGLE, vecAng );
        jointPoints -> setRotation( vecAng[3]/2, vecAng[X], vecAng[Y], vecAng[Z] );
    }
    return rotationAccepted;
}

//-----
void Segment::getOrientation( const ROTATION_METHODS method,
                             GLfloat * const orientation )
{
    localMatrix ->getOrientation( method, orientation );
}

//-----
boolean Segment::rotate( const ROTATION_METHODS method,
                        const GLfloat *orientation )
{
    boolean rotationAccepted =
        localMatrix ->rotate( method, orientation );
    //checks if rotation in boundaries
    if( rotationAccepted ){
        if( jointPoints != NULL ){

            GLfloat vecAng[THREE_D+1];
            localMatrix ->getOrientation( VECTOR_ANGLE, vecAng );
            jointPoints -> setRotation( vecAng[3]/2, vecAng[X], vecAng[Y], vecAng[Z] );
        }
    }
    return rotationAccepted;
}

```

```

//-----
void Segment::setModelType( MODEL_TYPE type )
{
    modelType =type;
}

//-----
void Segment::setStickShape( Box& box )
{
    *stick =box;
    return;
}

//-----
void Segment::getEarthMatrix( GLfloat h_matrix[16] )
{
    for( GLint i=0; i<16; i++){
        h_matrix[i] =modelViewMatrix[i];
    }
}

//-----
void Segment::getJointCenters(GLfloat &jointX, GLfloat &jointY, GLfloat &jointZ)
{
    static const GLint POS =12;

    jointX =modelViewMatrix[POS+X];
    jointY =modelViewMatrix[POS+Y];
    jointZ =modelViewMatrix[POS+Z];

    return;
}

//-----
GLfloat Segment::getLength()
{
    return stick ->getHeight();
}

```

```

*****  

// FILE: SensorSystem.h  

// DESCRIPTION: Combines [DUMA99] thesis program with human project  

//  

// NOTICE : Quaternion class in [DUMA99] is different than  

//           QuaternionR   class of this study  

*****  

#ifndef __SENSORSYSTEM_H__  

#define __SENSORSYSTEM_H__  

#include <GL/glut.h>  

#include "utility.h"  

#include "Human.h"  

#include "Qaef.h"  

#include "Quaternion.h"  

class SensorSystem{  

public :-----P U B L I C-----  

    //CONSTRUCTORS  

    SensorSystem( Human * man);    //default  

    SensorSystem( SensorSystem & ); //copy  

    //DESTRUCTOR  

    ~SensorSystem();  

    //FUNCTIONs  

    void start();  

    void trackSegment();  

private :-----P R I V A T E-----  

    //OPERATORS  

    SensorSystem& operator=( const SensorSystem & );  

    //FUNCTIONs  

    //OBJECTs  

    Human * human;  

    Quaternion qRotation;  

    Qaef *q1, *q2;  

    //VARIABLEs  

    GLfloat orientation[ THREE_D+1 ];  

};  

#endif

```

```

//*****
// FILE: SensorSystem.cpp
// DESCRIPTION:
//*****

#include "SensorSystem.h"

//-----
SensorSystem::SensorSystem(Human * man ){

    // INITIALIZE
    human =man;

    q1 =new Qaef (1,400,4);
    q2 =new Qaef (2,400,4);
}

//-----
SensorSystem::~SensorSystem(){

}

//-----
void SensorSystem::start()
{
    q1 ->start();
    q2 ->start();
}

//-----
void SensorSystem::trackSegment(){

    Quaternion vecAng = (q1 ->getResult()).toAxisAngles();

    orientation[X] =vecAng.getX();
    orientation[Y] =vecAng.getY();
    orientation[Z] =vecAng.getZ();
    orientation[3] =vecAng.getW();

    human -> setPosture( VECTOR_ANGLE, R_UPPER_ARM, orientation );

    vecAng = (q2 ->getResult()).toAxisAngles();

    orientation[X] =-vecAng.getX();
    orientation[Y] =vecAng.getY();
    orientation[Z] =-vecAng.getZ();
    orientation[3] =vecAng.getW();

    human -> setPosture( VECTOR_ANGLE, R_FORE_ARM, orientation );
}
}

```

```

//*****
// FILE    : TriangleFaceSet.h
// DESCRIPTION: Segment shapes are drawn and handled
//*****


#ifndef __TRIANGLEFACESET_H__
#define __TRIANGLEFACESET_H__


#include <GL/glut.h>
#include "Box.h"
#include "utility.h"

class TriangleFaceSet{

public :-----P U B L I C-----


    //CONSTRUCTORS
    TriangleFaceSet( const char *FILE_NAME, GLfloat ** points );
    TriangleFaceSet( TriangleFaceSet & );


    //DESTRUCTOR
    ~TriangleFaceSet();


    //FUNCTIONs
    void drawTriangles( GLfloat ** points, GLfloat ** normals );
    void showBounds();


    boolean isConstructed();


private :-----P R I V A T E-----


    //CONSTs
    static const GLfloat PORTION;


    //OPERATORS
    TriangleFaceSet& operator=( const TriangleFaceSet & );


    //FUNCTIONs
    void constructIndices( const char *FILE_NAME );
    void constructBounds( GLfloat ** points );


    //OBJECT POINTERS
    Box *bounds;


    //VARIABLEs
    GLint
        numberOfIndices, //number of points that draw this shape
        *indices;      //index numbers for points
    };


#endif

```

```

//*****
// FILE    : TriangleFaceSet.cpp
// DESCRIPTION:
//*****
#include "TriangleFaceSet.h"

//*****INITIALIZE STATIC DATA MEMBERS *****
const GLfloat TriangleFaceSet::PORTION =10;

//*****END STATIC DATA MEMBER INITIALIZATION *****

//-----
TriangleFaceSet::TriangleFaceSet( const char *FILE_NAME, GLfloat ** points  )
{
    // INITIALIZE
    indices =NULL;
    bounds =NULL;

    constructIndices( FILE_NAME );

    if( indices !=NULL ){

        constructBounds( points );
    }
}

//-----
TriangleFaceSet::~TriangleFaceSet()
{
    delete [] indices;
    delete bounds;
}

//-----
void TriangleFaceSet::constructIndices( const char *FILE_NAME )
{
    numberIndices =0;

    indices =readIndices( FILE_NAME, numberIndices );

    if( indices != NULL ){

        cout << " "
            << " TriangleFaceSet is constructed with "
            << ( numberIndices / 3 ) // each polygon has 3 points
            << " polygons from "
            << FILE_NAME
            << endl;
    }
}

return;
}

```

```

//-----
void TriangleFaceSet::constructBounds( GLfloat ** points ){

    const GLfloat MAX =1000;

    GLfloat minY =MAX, maxY =-MAX,
            volume[TWO_PLATES][FOUR_POINTS][THREE_D];

    // SET min-max Y values of all points of this segment
    for(GLint i =0; i<numberOfIndices; i++){

        minY =(minY < points[indices[i]][Y]) ? minY : points[indices[i]][Y];
        maxY =(maxY > points[indices[i]][Y]) ? maxY : points[indices[i]][Y];
    }

    // initialize VOLUME parameters
    for(GLint plate=0; plate<TWO_PLATES; plate++){

        volume[plate][0][X] =MAX;
        volume[plate][0][Z] =MAX;
        volume[plate][2][X] =-MAX;
        volume[plate][2][Z] =-MAX;
    }

    for(GLint point=0; point<FOUR_POINTS; point++) {

        volume[BOTTOM_PLATE][point][Y] =minY;
        volume[TOP_PLATE][point][Y] =maxY;
    }

    // find VOLUME values
    for(i =0; i<numberOfIndices; i++){

        for(GLint plate=0; plate<TWO_PLATES; plate++)
        for(GLint xyz=0; xyz<THREE_D; xyz++){

            if( xyz!= Y &&
                // check only for X & Z and if Y values in range length/PORTION
                ( // capture minimum and maximum values of X and Z for minimum Y
                  ( plate==BOTTOM_PLATE && points[indices[i]][Y]
                  < minY+((maxY-minY)/PORTION) )

                || // capture minimum and maximum values of X and Z for maximum Y

                  ( plate==TOP_PLATE && points[indices[i]][Y]
                  > maxY-((maxY-minY)/PORTION) )
                )
            ){

                volume[plate][0][xyz] // put min values in xyz0
                =(volume[plate][0][xyz] < points[indices[i]][xyz]) ?
                volume[plate][0][xyz] : points[indices[i]][xyz];

                volume[plate][2][xyz] // put max values in xyz2
                =(volume[plate][2][xyz] > points[indices[i]][xyz]) ?
                volume[plate][2][xyz] : points[indices[i]][xyz];
            }
        }
    }
}

```

```

        // put values from point0 & point2 into point1 & point3
        volume[plate][1][X] =volume[plate][0][X];
        volume[plate][1][Z] =volume[plate][2][Z];
        volume[plate][3][X] =volume[plate][2][X];
        volume[plate][3][Z] =volume[plate][0][Z];
    } // end if
} // end of for
} // end of for

bounds =new Box( volume );

return;
}

//-----
void TriangleFaceSet::drawTriangles( GLfloat ** points, GLfloat ** normals ){
    for(GLint i=0; i<numberOfIndices; i++){
        if( (i % 3) ==0 ) glBegin(GL_TRIANGLES);

        glNormal3fv( normals[indices[i]] );
        glVertex3fv( points[indices[i]] );

        if( ((i+1) % 3) ==0 ) glEnd();
    }

    return;
}

//-----
void TriangleFaceSet::showBounds()
{
    if( bounds != NULL ){
        bounds ->show();
    }

    return;
}

//-----
boolean TriangleFaceSet::isConstructed()
{
    boolean status =FALSE;

    if ( indices !=NULL ){
        status =TRUE;
    }

    return status;
}
}

```

```

***** :UserControl.h
// DESCRIPTION: Interface between GlutBaseClass and motion control classes
*****


#ifndef __USERCONTROL_H__
#define __USERCONTROL_H__


#include <GL/glut.h>
#include "utility.h"
#include "Human.h"
#include "Segment.h"
#include "Cursor3D.h"
#include "GimbalSystem.h"
#include "InverseKinematics.h"


//ENUMs
enum CONTROL_TYPE{ INVERSE_CONTROL, EULER_CONTROL, QUATERNION_CONTROL };
enum OBJECT_TYPE{ SEGMENT_SHAPES, EULER_CIRCLES };

class UserControl{

public :-----PUBLIC-----

    //CONSTRUCTORS
    UserControl( Human * );           //default
    UserControl( UserControl & ); //copy

    //DESTRUCTOR
    ~UserControl();

    //FUNCTIONs
    void setControlType( CONTROL_TYPE );

    void mouseDragAt( const GLint WIN_X, const GLint WIN_Y );
    void mouseReleasedAt( const GLint WIN_X, const GLint WIN_Y );
    void mouseHitAt( const GLint WIN_X, const GLint WIN_Y,
                     const GLfloat viewRotX, const GLfloat viewRotY);

    boolean isTracking();

private :-----PRIVATE-----

    //OPERATORS
    UserControl& operator=( const UserControl & );

    //FUNCTIONs
    GLuint isAnyObjectSelected( const GLint, const GLint, OBJECT_TYPE );
    void trackingEndAt ( const GLint WIN_X, const GLint WIN_Y );
    void trackingStartAt( const GLint WIN_X, const GLint WIN_Y,
                         const GLfloat viewRotX, const GLfloat viewRotY);
    void markVectorAngle( const GLint WIN_X, const GLint WIN_Y );
    void initializeGimbalSystem();
    void updateGimbalSystem(const GLint WIN_X, const GLint WIN_Y);

    void quaternionMotion(const GLint WIN_X, const GLint WIN_Y);
}

```

```
boolean isEndEffector( GLuint );

//OBJECT POINTERS
Human * human;
Cursor3D * selectionMark, * cursor3D;
GimbalSystem gimbal;
InverseKinematics *inverseK;

//VARIABLEs
GLint oldWinX, oldWinY;

GLfloat
selectedSegment_jx,
selectedSegment_jy,
selectedSegment_jz,
orientation[THREE_D+1];

SEGMENTS selectedSegment;

AXIS selectedAxis;

CONTROL_TYPE controlType;
};

#endif
```

```

//*****
// FILE    : UserControl.cpp
// DESCRIPTION:
//*****

#include < math.h >
#include "UserControl.h"

//-----
UserControl::UserControl( Human * man )
{
    // INITIALIZE
    controlType      =QUATERNION_CONTROL;
    human           =man;
    selectedSegment =NONE;
    selectedAxis    =UNDEFAXIS;

    selectionMark =new Cursor3D();
    cursor3D     =new Cursor3D();

    inverseK =new InverseKinematics ( man );
}

//-----
UserControl::~UserControl()
{
    delete cursor3D;
    delete selectionMark;
}

//-----
void UserControl::mouseHitAt(const GLint WIN_X, const GLint WIN_Y,
                           const GLfloat viewRotX, const GLfloat viewRotY)
{
    switch( controlType ){

        case INVERSE_CONTROL :
            //same with quat
        case QUATERNION_CONTROL:
            trackingStartAt( WIN_X, WIN_Y, viewRotX, viewRotY);
            break;
        case EULER_CONTROL :{

            GLuint selection =isAnyObjectSelected( WIN_X, WIN_Y, EULER_CIRCLES );

            if(selection ==NO_SELECTION){

                selection =isAnyObjectSelected( WIN_X, WIN_Y, SEGMENT_SHAPES );
                if( selection!=NO_SELECTION &&
                    selectedSegment==((SEGMENTS) selection )){

                    //switch to QUATERNION_CONTROL control
                    selectedSegment =NONE;
                    controlType =QUATERNION_CONTROL;
                }
            }
        }
    }
}

```

```

        else{
            selectedAxis =(AXIS) selection;
        }

    }
    break;
default: break;
}

oldWinX =WIN_X;
oldWinY =WIN_Y;
}

//-----
void UserControl::mouseReleasedAt(const GLint WIN_X, const GLint WIN_Y)
{
    if( selectedSegment != NONE ){

        switch( controlType ){
            case INVERSE_CONTROL :
                //same with quat
            case QUATERNION_CONTROL:

                if( WIN_X==oldWinX && WIN_Y==oldWinY ){

                    //switch to EULER_CONTROL control
                    controlType =EULER_CONTROL;
                    initializeGimbalSystem();
                }
                else{
                    trackingEndAt( WIN_X, WIN_Y );
                }
                break;
            case EULER_CONTROL :
                selectedAxis =UNDEFAXIS;
                break;
            default: break;

        }//end switch
    }//end if
}

//-----
void UserControl::mouseDragAt(const GLint WIN_X, const GLint WIN_Y)
{
    switch( controlType ){

        case INVERSE_CONTROL :
            //same with quat
        case QUATERNION_CONTROL:
            markVectorAngle( WIN_X, WIN_Y );
            break;
        case EULER_CONTROL :
            updateGimbalSystem( WIN_X, WIN_Y );
            //hold mouse position
            oldWinX =WIN_X;
    }
}

```

```

        oldWinY =WIN_Y;
        break;
    default: break;
}
}

//-----
void UserControl::initializeGimbalSystem()
{
    human->getPosture( EULER, selectedSegment, orientation );
    gimbal.setAngle( orientation );
}

//-----
void UserControl::updateGimbalSystem(const GLint WIN_X, const GLint WIN_Y)
{
    if( selectedAxis !=UNDEFAXIS ) {

        if(WIN_Y-oldWinY > 0 || WIN_X-oldWinX > 0){

            gimbal.increment( selectedAxis );
        }
        else if(WIN_Y-oldWinY < 0 || WIN_X-oldWinX < 0){

            gimbal.decrement( selectedAxis );
        }
        gimbal.getAngle( orientation );
        if( ! human ->modifyPosture( EULER, selectedSegment, orientation ) ){

            initializeGimbalSystem();
        }
    }
    //draw GimbalSystem
    glTranslatef( -3, 3, -10);
    glScalef( 0.1f, 0.1f, 0.1f );
    gimbal.draw();
    glLoadIdentity();
}
}

//-----
void UserControl::trackingStartAt(const GLint WIN_X, const GLint WIN_Y,
                                const GLfloat viewRotX, const GLfloat viewRotY)
{
    // check whether mouse is on human obj
    GLuint selection =isAnyObjectSelected( WIN_X, WIN_Y, SEGMENT_SHAPES );

    if(selection ==NO_SELECTION){

        selectedSegment =NONE;
    }
    else if( controlType ==QUATERNION_CONTROL || isEndEffector(selection) ) {

        inverseK ->setEarthOrientation(viewRotX, viewRotY);
        selectedSegment =(SEGMENTS) selection;
    }
}

```

```

if( selectedSegment != NONE ){

    // gets joint centers of selected segment
    human ->getJointCenters(selectedSegment,
                                selectedSegment_jx,
                                selectedSegment_jy,
                                selectedSegment_jz );

    cursor3D ->setWorldCoord( 0, 0, selectedSegment_jz );
    if( controlType ==QUATERNION_CONTROL ){

        selectionMark ->setWorldCoord( 0, 0, selectedSegment_jz );
        selectionMark ->setWindowCoord( WIN_X, WIN_Y );

        GLdouble
        cx =cos( -viewRotX * DEG_TO_RAD ),
        sx =sin( -viewRotX * DEG_TO_RAD ),
        cy =cos( -viewRotY * DEG_TO_RAD ),
        sy =sin( -viewRotY * DEG_TO_RAD );

        // const. rot. vector from study angle
        orientation[X] =(GLfloat) ( cx*sy );
        orientation[Y] =(GLfloat) ( -sx );
        orientation[Z] =(GLfloat) ( cx*cy );
    }
}

//-----
boolean UserControl::isEndEffector( GLuint selection )
{
    SEGMENTS segment =(SEGMENTS) selection;
    boolean result =false;

    if( segment ==L_HAND || segment ==R_HAND ||
        segment ==L FOOT || segment ==R FOOT )
    {
        inverseK ->initialize( segment );
        result =true;
    }

    return result;
}

//-----
void UserControl::trackingEndAt(const GLint WIN_X, const GLint WIN_Y)
{
    if( selectedSegment != NONE ){
        //convert window coord of mouse to world coord
        glLoadIdentity(); // Reset the modelview matrix
        cursor3D ->setWindowCoord( WIN_X, WIN_Y );

        if( controlType ==QUATERNION_CONTROL ){

            quaternionMotion(WIN_X, WIN_Y);
        }
    }
}

```

```

        else{
            inverseK ->algebraicSolution( (GLfloat)cursor3D ->getWorldX(),
                                            (GLfloat)cursor3D ->getWorldY(),
                                            selectedSegment_jz);
        }
        // turn selected option off
        selectedSegment =NONE;
    }
}

//-----
void UserControl::quaternionMotion(const GLint WIN_X, const GLint WIN_Y)
{
    GLfloat
    oldX =(GLfloat)selectionMark ->getWorldX(),
            oldY =(GLfloat)selectionMark ->getWorldY(),
            newX =(GLfloat)cursor3D ->getWorldX(),
            newY =(GLfloat)cursor3D ->getWorldY();

    // get angle between selected mouse coord. & joint & current mouse coord
    orientation[3] =getAngleFm3Points(selectedSegment_jx,selectedSegment_jy,
                                      oldX, oldY,
                                      newX, newY);

    // rotate segment with this rot. ang. & vec.
    human ->modifyPosture( VECTOR_ANGLE, selectedSegment, orientation );
}

//-----
void UserControl::markVectorAngle( const GLint WIN_X, const GLint WIN_Y )
{
    if ( selectedSegment != NONE ){ // check if a selection occured.
        // If it did, draw markers
        cursor3D ->setWindowCoord( WIN_X, WIN_Y );

        glDisable(GL_LIGHTING);
        glLineWidth(3);
        //draw cursors
        cursor3D ->draw();

        if( controlType ==QUATERNION_CONTROL ){
            selectionMark ->draw();

            //draw triangle (marking rotation angle)
            glColor3f(1,1,1);
            glBegin( GL_LINE_LOOP );
                glVertex3d( selectionMark ->getWorldX(),
                            selectionMark ->getWorldY(),
                            selectionMark ->getWorldZ() );
                glVertex3f( selectedSegment_jx,
                            selectedSegment_jy,
                            selectedSegment_jz );
                glVertex3d( cursor3D ->getWorldX(),
                            cursor3D ->getWorldY(),
                            cursor3D ->getWorldZ() );
            glEnd();
        }
    }
}

```

```

        else{
            //draw triangle (marking rotation angle)
            glColor3f(1,1,1);
            glBegin( GL_LINES );
                glVertex3f( selectedSegment_jx,
                            selectedSegment_jy,
                            selectedSegment_jz );
            glVertex3d( cursor3D ->getWorldX(),
                        cursor3D ->getWorldY(),
                        cursor3D ->getWorldZ() );
            glEnd();
        }

        glLineWidth(1);

        glEnable(GL_LIGHTING);
    }

//-----
void UserControl::setControlType( CONTROL_TYPE type )
{
    controlType =type;
}

//-----
boolean UserControl::isTracking()
{
    return ( (selectedSegment == NONE) ? FALSE : TRUE );
}

//-----
GLuint UserControl::isAnyObjectSelected(const GLint mouseX, const GLint mouseY,
                                         OBJECT_TYPE objectType )
{
    GLuint selectedObject =NO_SELECTION;

    GLfloat maxZ =0;
    GLuint selectBuffer[64] ={ 0 };
    GLint hits =0, viewPort[4];

    glSelectBuffer(64, selectBuffer); //init. select buf.

    glGetIntegerv(GL_VIEWPORT, viewPort);

    glLoadIdentity(); // Reset the modelview matrix
    glMatrixMode(GL_PROJECTION); //change matrix mode to project
    glPushMatrix();

    glRenderMode(GL_SELECT); //change render mode to SELECT
    glLoadIdentity(); //clear matrix

    //track a view volume for detection
    gluPickMatrix( mouseX, viewPort[3]-mouseY, 2, 2, viewPort );
    gluPerspective(45,(GLfloat)viewPort[2]/(GLfloat)viewPort[3],1,50);
}

```

```

glMatrixMode(GL_MODELVIEW);

//draw shapes as detectors
if ( objectType == SEGMENT_SHAPES ){
    human->drawMouseDetectors();
}
else{
    glLoadIdentity();
    glTranslatef( -3, 3, -10);
    glScalef( 0.1f, 0.1f, 0.1f );
    gimb.al.drawMouseDetectors();
}

glLoadIdentity(); // Reset the modelview matrix
glMatrixMode(GL_PROJECTION);

hits =glRenderMode(GL_RENDER); //if any drawn shape hit to viewing volume
//get number of hit objects

if( hits > 0 ) { //if there are objects which is hit

    GLint hitObj =0;
    for(GLint i=0; i<hits; i++){ // choose the one which has max-Z

        if(selectBuffer[(i*4)+2]<selectBuffer[(hitObj*4)+2]) hitObj =i;
    }
    // set selection
    selectedObject =selectBuffer[(hitObj*4)+3];
}

glMatrixMode(GL_PROJECTION); //set project
glPopMatrix();
glMatrixMode(GL_MODELVIEW); // change matrix mode to model

return selectedObject;
}

```

```

//***** : Utility.h
// DESCRIPTION: general purpose functions
//***** 

#ifndef UTILITY_H
#define UTILITY_H

#include <GL/glut.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>

#ifndef NULL
#define NULL 0
#endif

// ENUMS
enum AXIS{ X, Y, Z, XY, XZ, YZ, XYZ, UNDEFAXIS };// axis names

enum ROTATION_METHODS{ QUATERNION, EULER, VECTOR_ANGLE, MATRIX };

// CONSTANTS

// max file name must be equal or greater than actual file names.
const GLbyte MAX_FILE_NAME = 16;

// symbol for 3
const GLbyte THREE_D =3;

// symbol for 2
const GLbyte TWO_D =2;

// degree to radian
const GLfloat DEG_TO_RAD =0.01745f;

const GLuint NO_SELECTION =-1;

const GLfloat OUT_RANGE =361.0f;

// FUNCTIONS

// reads integers from the given file to returned int array
GLint* readIndices( const char *const, GLint &);

// reads floats from the given file to a returned float array
GLfloat** readPoints( const char *const INFILE_NAME, GLint &numberOfPoints );

// returns angle which is between line (x1,y1)-(cx,cy) and
// line (x2,y2)-(cx,cy)
GLfloat getAngleFrom3Points(const GLfloat, const GLfloat,
                           const GLfloat, const GLfloat,
                           const GLfloat, const GLfloat);

```

```
void twoLink2D(const GLfloat, const GLfloat,
               const GLfloat, const GLfloat,
               GLfloat &,   GLfloat &           );
GLfloat linearInterpolate( const GLfloat VALUE,
                           const GLfloat TIME, const GLfloat CURRENT_TIME );
void quatInterpolation( const GLfloat * const, const GLfloat * const,
                        const GLfloat, GLfloat * const );
#endif
```

```

//*****:utility.cpp
// DESCRIPTION:
//*****:utility.h

#include "utility.h"

//***** AUX. UTILIY FUNC. that are used in this file ***

// utility function that open an input file.Returns TRUE , if successful
boolean openInputFile( ifstream & , const char *const );

// returns the length between (x1,y1) and (x2,y2)
GLfloat length(const GLfloat, const GLfloat,
                const GLfloat, const GLfloat);

// returns the angle by cos theorem for given 3 points
GLfloat cosTheory( const GLfloat, const GLfloat, const GLfloat);

//***** UTILIY FUNCTIONS *****
//-----
boolean openInputFile( ifstream &inputFile, const char *const FILE_NAME ){

    boolean returnValue = TRUE ; // return value of this function

    inputFile.open( FILE_NAME, ios::in );

    if( !(inputFile) ){

        cout << "Input File open falsed."
            << "Can't open " << FILE_NAME << endl;

        returnValue = FALSE;
    }
    return returnValue;
}

//-----
GLint* readIndices( const char *const INFILE_NAME, GLint & numberOfIndices ){

    ifstream inFile; // input file
    GLint *indices =NULL; // return array

    if ( openInputFile( inFile, INFILE_NAME )==TRUE ){
        // file is opened successfully
        GLint value;
        numberofIndices =-1;

        // count ints in the file
        while( !(inFile.eof() ) ){

            inFile >> value ;
            ++numberofIndices;
        };
    }
}

```

```

// turn pointer to the beginning of the file
inFile.clear();
inFile.seekg(0);

//initialize memory for return array
indices =new GLint[numberOfIndices];

// read file into return array
for( GLint i=0; i<numberOfIndices; i++){

    inFile >> indices[i];
}

// close file
inFile.close();
}

return indices;
}

//-----
GLfloat** readPoints( const char *const INFILE_NAME, GLint &numberOfPoints ){

ifstream inFile; // input file
GLfloat **points =NULL;

if ( openInputFile( inFile, INFILE_NAME ) ==TRUE ){
    // file is opened successfully
    GLfloat value;
    numberOfPoints =-1;

    // count floats in the file
    while( !( inFile.eof() ) ){

        inFile >> value ;
        ++numberOfPoints;
    };

    // turn pointer to the begining of the file
    inFile.clear();
    inFile.seekg(0);

    //initialize memory for return array
    points =new GLfloat* [numberOfPoints];

    // read file into return array
    for( GLint i=0; i<numberOfPoints; i++){

        points[i] =new GLfloat[THREE_D];
        for( GLint j=0; j<THREE_D; j++){

            inFile >> points[i][j];
        }
    }
}

```

```

// close file
inFile.close();
}

return points;
}

//-----
GLfloat getAngleFrom3Points(const GLfloat centerX, const GLfloat centerY,
                           const GLfloat x1, const GLfloat y1,
                           const GLfloat x2, const GLfloat y2)
{
    GLfloat angle, angle1, angle2;

    angle1 = cosTheory(length(centerX,centerY+1,x1, y1),
                       length(centerX,centerY,x1, y1),
                       1 );
    angle2 = cosTheory(length(centerX,centerY+1,x2, y2),
                       length(centerX,centerY,x2, y2),
                       1 );

    if( x1>=centerX && x2>=centerX ){

        angle =angle1-angle2;
    }
    else if( x1<=centerX && x2<=centerX ){

        angle =angle2-angle1;
    }
    else if( x1>centerX && x2<centerX ){

        angle =angle1+angle2;
        if(angle>180) angle-=360;
    }
    else if( x1<centerX && x2>centerX ){

        angle =-1*(angle1+angle2);
        if(angle<-180) angle+=360;
    }

    return angle;
}

//-----
GLfloat length( const GLfloat x1, const GLfloat y1,
                 const GLfloat x2, const GLfloat y2)
{
    GLfloat dx=x1-x2,
            dy=y1-y2;

    return (GLfloat) sqrt((dx*dx)+(dy*dy));
}

```

```

//-----
GLfloat cosTheory( const GLfloat a, const GLfloat b, const GLfloat c)
{
    GLfloat angle=acos(
        ((a*a)-(b*b)-(c*c))
        /
        (-2 * b * c )
    );

    return ( angle / DEG_TO_RAD );
}

//-----
void twoLink2D(const GLfloat X_POS,  const GLfloat Y_POS,
               const GLfloat LENGTH_1, const GLfloat LENGTH_2,
               GLfloat & teta1,      GLfloat & teta2      )
{
    //calc. of link angles to give posture to system
    teta2 =acos(
        (
            (X_POS*X_POS)
            + (Y_POS*Y_POS)
            - (LENGTH_1*LENGTH_1)
            - (LENGTH_2*LENGTH_2)
        )
        /
        ( 2 * LENGTH_1 * LENGTH_2 )
    );

    teta1 =atan( X_POS/Y_POS )
        - atan(
            ( LENGTH_2 * sin(teta2) )
            /
            ( LENGTH_1 + ( LENGTH_2*cos(teta2) ) )
        );

    //convert angles to rad
    teta1/=DEG_TO_RAD;
    teta2/=DEG_TO_RAD;

    return;
}

//-----
GLfloat linearInterpolate( const GLfloat VALUE,
                           const GLfloat TIME, const GLfloat CURRENT_TIME )
{
    return ( CURRENT_TIME * VALUE / TIME );
}

```

```

//-----
void quatInterpolation( const GLfloat * const q1,
                        const GLfloat * const q2,
                        const GLfloat t, GLfloat * const newQ )
{
    static const GLfloat HALF_PI =1.570796f;

    if( q1[X]==q2[X] && q1[Y]==q2[Y] && q1[Z]==q2[Z] && q1[3]==q2[3] ){

        for( GLint i=0; i<4; i++ ){ newQ[i] =q1[i]; }

    } else{
        GLfloat cosOm, omega, sinOm, sclp, sclq;

        cosOm =(q1[X]*q2[X]) + (q1[Y]*q2[Y]) + (q1[Z]*q2[Z]) + (q1[3]*q2[3]);

        if( cosOm != -1 ){ //angle between q1 and q2 < 180

            if( cosOm != 1 ){ //180 > angle > 0

                omega =( GLfloat )acos( cosOm );
                sinOm =( GLfloat )sin( omega );
                sclp =( GLfloat )sin( (1-t)*omega )/sinOm;
                sclq =( GLfloat )sin ( t*omega )/sinOm;

            }

            else{ //angle =0
                sclp =1-t;
                sclq =t;
            }

        }

        for( GLint i=0; i<4; i++ ){ newQ[i] =(q1[i]*sclp) + (q2[i]*sclq); }

    } else{ //q1 is opposite to q2.Angle =180;

        newQ[X] =-q1[Y];
        newQ[Y] =q1[X];
        newQ[Z] =-q1[3];
        newQ[3] =q1[Z];

        sclp =( GLfloat )sin( (1-t)*HALF_PI );
        sclq =( GLfloat )sin( t*HALF_PI );

        for( GLint i=0; i<4; i++ ){ newQ[i] =(q1[i]*sclp) + (newQ[i]*sclq); }

    }

}
}

```

LIST OF REFERENCES

- [BADL93a] Badler, N. I., Phillips, C. B. and Webber, B. L., "Simulating Humans: Computer Graphics Animation and Control", Oxford University Press, New York, 1993.
- [BADL93b] Badler, N. I., Hollick, M. J. and Granieri, J. P., "Real-Time Control of a Virtual Human Using Minimal Sensors," *Presence: Teleoperators and Virtual Environments*, Winter 1993, Volume 2, Number 1, pp. 82-86.
- [BEDI97] Bediz, Mehmet "A Computer Simulation Study of a Single Rigid Body Dynamic Model for Biped Postural Control" Master's Thesis, Naval Postgraduate School, Monterey, California, March 1997.
- [COOK92] Cooke, Joseph M., Zyda, Michael J., Pratt, David R., and McGhee, Robert B., "NPSNET: Flight Simulation Dynamic Modeling Using Quaternions," *Presence*, Fall 1992, Volume 1, Number 4, pp. 405-420.
- [CRAI89] Craig, J. John, "Introduction to Robotics, Mechanics and Control", Addison-Wesley Publishing Company, Inc.1989, Second Edition.
- [DAVI93] Davidson, Sandra L., "An Experimental Comparison of CLOS and C++ Implementations of an Object-Oriented Graphical Simulation of Walking Robot Kinematics", Master's Thesis, Naval Postgraduate School, Monterey, California, March 1993.
- [DUMA99] Duman, Ildeniz, "Implementation and Testing of a Real-Time Software System for a Quaternion-Based Attitude Estimation Filter", Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1999.
- [DURL95] Durlach, N. I. and Mavor, A. S., National Research Council, "Virtual Reality: Scientific and Technological Challenges", National Academy Press, Washington, D.C., 1995, pp. 188-204, 306-317.
- [FREY96] Frey, William, III, "Application of Inertial Sensors and Flux-Gate Magnetometer to Real-Time Human Body Motion Capture", Master's Thesis, Naval Postgraduate School, Monterey, California, September 1996.
- [FUND90] Funda, J., Taylor, R., PAUL, R. P., "On Homogenous Transform, Quaternions, and Computational," *IEEE Transactions On Robotics and Automation*, Vol.6, No. 3, 1990.
- [GOET94] Goetz, John Robert, "Graphical Simulation of Articulated Rigid Body System Kinematics with Collision Detection", Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

[GRAN95] Granieri, J. P. and Badler, N. I., "Simulating Humans in VR," *Virtual Reality Applications*, Academic Press, ISBN 0-12-227755-4, 1995, pp. 253-269.

[HODG95] Hodgins, J. K., Wooten, W. L., Brogan, D. C., O'Brien, J. F., "Animating Human Athletics," *Proceedings of SIGGRAPH '95*, Los Angeles, CA, August 6-11, In *Computer Graphics*, 1995, pp 71-78.

[KOOZ83] Koozekanani, S. H., Barin, K., McGhee R. B., Chang, H. T., "A Recursive Free-Body Approach to Computer Simulation of Human Postural Dynamics", *IEEE Transactions on Biomedical Engineering*, Vol. BME-30, No. 12, December 1983, pp.788-789.

[MCGH79] McGhee, R. B., "Computer Simulation of Human Movement", *CISM Courses and Lectures No. 263*, International Center for Mechanical Sciences, Springer-Verlag Wien-New York, 1980.

[MCGH86] McGhee, R. B., Nakano, E., Koyachi, N., Adachi, H., "An Approach to Computer Coordination of Motion for Energy-Efficient Walking Machines", *Bulletin of Mechanical Engineering Laboratory, JAPAN*, Number 43, 1986.

[PRAT93] Pratt, David R., "A Software Architecture for the Construction and Management of Real-Time Virtual Worlds," *Dissertation*, Naval Postgraduate School, Monterey, CA, June 1993.

[PRAT94] Pratt, D. R., Barham, P. T., Locke, J., Zyda, M. J., Eastman, B., Moore, T., Biggers, K., Douglass, R., Jacobsen, S., Hollick, M., Granieri, J., Ko, H. and Badler, N. I., "Insertion of an Articulated Human into a Networked Virtual Environment," *Proceedings of the Fifth Annual Conference on AI, Simulation and Planning in High Autonomy Systems: Distributed Interactive Simulation Environments*, IEEE Computer Society Press, Gainesville, Florida, December 7-9,1994, pp. 84-90.

[PRAT95] Pratt, S. M., Pratt, D. R., Waldrop, M. S., Barham, P. T., Ehlert, J. F. and Chrislip, C. A., "Humans in a Large-Scale, Real Time, Networked Virtual Environments," submitted to *Presence*, 1996.

[SKOP97] Skopowski, Paul F. "Immersive Articulation of the Human Upper Body in a Virtual Environment" *Master's Thesis*, Naval Postgraduate School, Monterey, California, January 1997.

[WALD95] Waldrop, Marianne S., "Real-time Articulation of the Upper Body for Simulated Humans in Virtual Environments", *Master's Thesis*, Naval Postgraduate School, Monterey, California, September, 1995.

[WATT92] Watt, A. and Watt, M., "Advanced Animation and Rendering Techniques: Theory and Practice", Addison-Wesley Publishing Company, Inc., New York, 1992, pp. 369-394.

[WEBREF1] Images from Geri's Game, 1997 (Pixar Animation Studios).
<http://www.pixar.com/shorts/geri/geri.html>

[WEBREF2] ULTRATRAK PRO from Polhemus.
<http://www.polhemus.com/trackers>

[WEBREF3] VRML file that demonstrates Gimbal system
<http://www-npsnet.cs.nps.navy.mil/bachmann/orientation/orientation.wrl>

[WEBREF4] VRML file of a human body
<http://www.ocnus.com/models>

[WAVE98] Alias | Wavefront inc., Learning Maya, 1998.

[ZYDA92] Zyda, M. J., Pratt, D. R., Monahan, J. G. and Wilson, K. P., "NPSNET: Constructing a 3D Virtual World," 1992 Proceedings of Symposium on Interactive 3D Graphics, pp. 147-156.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060-6218
2. Dudley Knox Library.....2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101
3. Chairman, Code CS1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000
4. Dr. Robert B. McGhee, Professor.....2
Computer Science Department Code CS/MZ
Naval Postgraduate School
Monterey, CA 93943-5000
5. Dr. Michael J. Zyda, Professor2
Computer Science Department Code CS/ZK
Naval Postgraduate School
Monterey, CA 93943-5000
6. LT(jg) Umit Y. Usta.....2
Dumlupinar mah.
Prevezе cad. No: 84
Golcuk / KOCAELI
Turkey
7. Deniz Kuvvetleri Komutanligi1
Personel Tedarik ve Yetistirme Daire Baskanligi
Bakanliklar, Ankara 06100
Turkey
8. METU (ODTU)1
06531 Ankara
Turkey

9. Bogazici University 1
80815 Bebek/Istanbul
Turkey

10. Bilkent University..... 1
Department of Computer Engineering and Information Science
06533 Bilkent/Ankara
Turkey